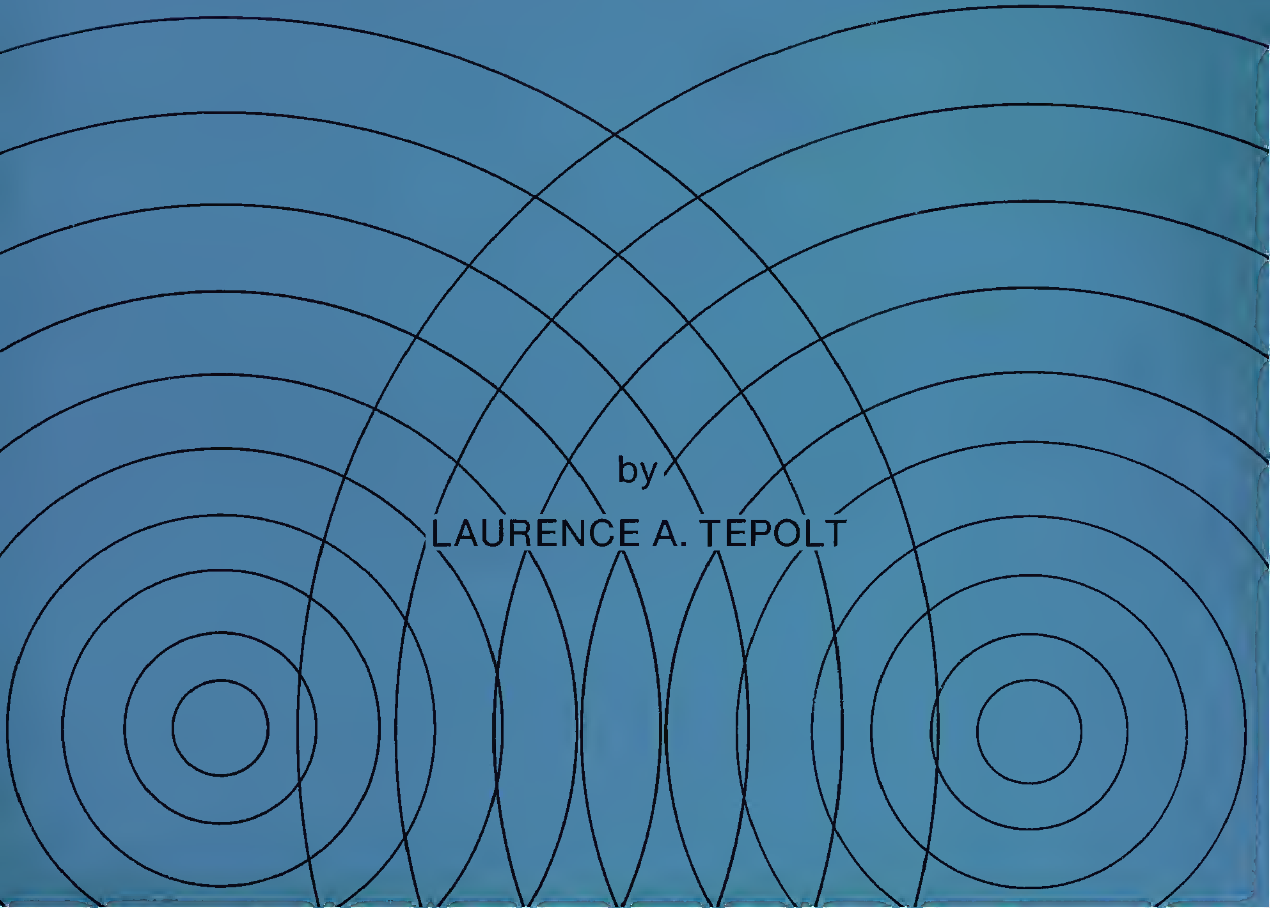
# Assembly Language Programming

for the

TRS-80® COLOR COMPUTER



by

LAURENCE A. TEPOLT

# ASSEMBLY LANGUAGE PROGRAMMING For The TRS-80 r COLOR COMPUTER plus a users guide to EDTASM+ r

By
Laurence A. Tepolt

Published by TEPCO
30 Water Street
Portsmouth, RI 02871

## Dedication

To my wife:
Janet Milotte Tepolt.

## Acknowledgments

I wish to thank the many people who encouraged me to write this book, and those who allowed me to devote most of my personal time to this effort. Special thanks to Motorola, Inc., who allow some of their documentation to be reprinted here. Finally, I am indebted to Donald R. Milotte for his comments, suggestions, and his careful review as the book was being written.

# Preface

After happily owning and using my Color Computer for some time, I decided to try assembly language programming. I found no book available, however, that adequately presents assembly language programming applied specifically to the Color Computer. I ended up using several books, jumping from one to another to piece together the information I required for each program.

I became quite tired of not having the necessary information required to program my computer available in one place or book. So I began this book; a book for the assembly language programmer of the Color Computer. I structured this book as a learning aid for readers unfamiliar with assembly language so that they can learn it themselves. You will truly enjoy learning, and then programming, in assembly language; and you will be amazed at the results you can achieve on your Color Computer.

# Introduction

This book is for the Color Computer owner who has learned how to program in BASIC and wants to learn assembly language. Information and concepts are presented in an order that will facilitate learning: we'll progress from simple, more basic concepts in the early chapters through more advanced topics in later chapters. When new technical words are introduced their first use will be printed bold to get your attention. Specific information in this book applies to Color BASIC 1.1, Extended Color BASIC 1.0, Disk Extended Color BASIC 1.0, and the E revision Color Computer. Some of the differences (they are quite minor) in the Color Computer 2 are also presented.

Each new concept is followed by an example or two. Each example should be studied, although it would be better to work them out. Even better, try to construct similar examples to solve. You'll remember each concept more clearly if you work out some problems using them.

An assembly language programmer specifies each operation the computer will perform. Thus, this language is quite tedious. The computer will do just what you direct it, and nothing else, resulting in programs that run very fast. Assembly language is a **symbolic** language, in that the programmer specifies the operations to be performed by using their symbol. The programmer must understand the operations the computer can perform.

Chapter 1 is a comprehensive introduction to binary numbers, which all digital computers use. Digital computers work with digits, or numbers, and the numbering system they use is called the binary number system. Chapter 2 describes how binary numbers are used to represent information, as numbers or text data. Also covered are the operation and use of a computer's memory, and how information is organized in it. At the end of Chapter 2 I describe how Extended Color BASIC represents and stores data in memory.

Chapter 3 describes the MC6809E - the microprocessor used in the Color Computer. Its operation and internal structure are described in detail - neccessary information for an assembly language programmer. Chapters 4 and 5 describe the addressing modes and the instructions the MC6809E uses. Addressing modes are the various techniques the microprocessor uses to store data in or read data from memory. The instructions are the commands the microprocessor is capable of performing.

Chapter 6 describes the operation and use of EDTASM+, a very good text editor and assembler sold by Radio Shack. All assembly program examples in this book were written using the EDTASM+ ROM pack. You should understand as much as possible about the assembler you are using because it is the primary tool used to create a program. Chapter 7 describes how to write assembly language programs. Conventions and guidelines are provided to help the programmer write programs that work and that make editing or debugging much easier. Techniques for writing subroutines and interrupt handlers are also presented. Chapter 8 contains techniques and information needed to make assembly language programs work with BASIC programs. Many of the subroutines that exist in BASIC ROM are presented along with how your assembly language programs can use them. These subroutines primarily serve to transfer data between the computer and other devices such as the keyboard, screen, printer, joysticks, cassette, and disk.

Chapter 9 reveals other internal workings of the Color Computer. You will see how to control the graphics display modes, interrupts, and some other aspects of the computer's operation. Chapter 10 goes deeper into the operation of the computer, describing how data is transmitted to or received from other devices such as the printer, joysticks, keyboard, and cassette, and how to make and control sound. The cartridge connector is also described.

Even though much information is provided in this book, it is still an introduction. This book presents all the information about the Color Computer hardware any programmer might need. However, there are many more concepts and philosophies concerning assembly language programming than those presented here. For instance, many employers have established unique conventions and guidelines for programming that apply to specific types of programs that employer develops.

To use this book, start at the first chapter and procede to the last. Those already familiar with assembly language may start with Chapter 3. You will find this book provides all the information a beginner or expert needs to program the Color Computer in assembly language.

# Table of Contents

# CHAPTER 1

# The Binary Number System

Electronic digital computers are composed of tens of thousands of electronic circuits each of which must be economical to produce and also must operate reliably. To accomplish this, electronic circuits have been designed as electronic switches with two states: on and off. An electronic switch can be only on or off; this promotes the use of a number system with only two digits. This is the **binary** number system. Binary uses only two digits, 0 and 1, that respectively represent the two switch states, off and on.

The number system we are most familiar with is the decimal number system, made up of 10 digits, 0 - 9. Another way to describe the decimal number system is to say that it has a base, or **radix**, of 10. The base designation tells us the number of digits available in the system. A decimal number may be represented as:

$$235_{10}$$

where the subscript 10 indicates the base, or radix, of the number. In base 10, this is the familiar number 235; we don't print the base subsript because we assume everyone is using same base. However, when not using base 10, you should indicate this with a subscripted number that indicates the base being used.

The value of a number is the sum of all the products of each digit and the weight of that digit's position. In the above example, the 5 is in the units (one's) position, the 3 is in the tens position, and the 2 is in the hundreds position. Therefore, the value is calculated as:

$$235_{10} = (2x100)+(3x10)+(5x1)$$

In the following example there is a decimal point, also known as a radix point.

$$12.75_{10} = (1x10)+(2x1)+(7x.1)+(5x.01)$$

**Determining Position Weight**

Digit positions are counted in decimal, using the radix point as the starting point. The digit positions to the left of the radix point are counted by starting with position number zero (0) just to the left of the radix point. Each succeeding position number, proceding to the left from the radix point, is larger by one. Digit positions to the right of the radix point are counted by starting with position number minus one (-1), just to the right of the radix point. Each succeeding position number, to the right from the radix point, is decreased by one. The form used to indicate the digit positions of a numbering system in any base is:

$$\ldots\ 3\ 2\ 1\ 0\ .\ -1\ -2\ -3\ -4\ \ldots$$

where ... indicates an indefinite continuation.

Using the example number 12.75, the 1 is in digit position one (1), the 2 is in digit position zero (0), the 7 is in digit position minus one (-1), and the 5 is in digit position minus two (-2). The decimal weight of each digit position can be calculated now that we know the base and the digit position. The weight of a digit position is the base raised to the power of that position. In general, the digit position weights are calculated as follows:

$$\ldots\ b^2\ \ b^1\ \ b^0\ .\ b^{-1}\ \ b^{-2}\ \ b^{-3}\ \ldots$$

where b is the base of the numbering system. Therefore, the weights of each digit position for decimal numbers (b = 10) are:

$$\ldots\ 10^2\ \ 10^1\ \ 10^0\ .\ 10^{-1}\ \ 10^{-2}\ \ 10^{-3}\ \ldots$$

or

$$\ldots\ 100\ \ 10\ \ 1\ .\ 0.1\ \ .01\ \ .001\ \ldots$$

Raising a number to a power is accomplished by multiplying that number by itself the number of times the power, or exponent, indicates. If the exponent is negative, the result will be inverted. Examples of this procedure follow:

$$10^2 = 10x10 = 100$$
$$2^3 = 2x2x2 = 8$$
$$10^{-2} = 1/10^2 = 1/100 = .01$$

Two situations require further explanation. When a number (N) is raised to the power of one, the result is that number (N). For example:

$$N^1 = N,\ 10^1 = 10,\ 2^1 = 2$$

When any number is raised to the power of zero, the result is always one. For example:

$$N^0 = 1,\ 5^0 = 1,\ 2^0 = 1$$

**BINARY NUMBERS**

Since decimal and binary numbers will be presented concurrently, a convention must be established to differentiate one from the other. Each number will be referred to as decimal or binary or its base will be presented as a subscript. The binary number system has a base of two; the decimal weight of each digit position is calculated as shown:

$$\ldots\ 2^3\ \ 2^2\ \ 2^1\ \ 2^0\ .\ 2^{-1}\ \ 2^{-2}\ \ 2^{-3}\ \ 2^{-4}\ \ldots$$

where the exponent indicates the digit position.

| Digit Position | Decimal Weight |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 10 | 1024 |
| 11 | 2048 |
| 12 | 4096 |
| 13 | 8192 |
| 14 | 16384 |
| 15 | 32768 |

Table 1-1 Decimal Weights of Binary Digit Positions.

This may also be represented as:

... 8 4 2 1 . 1/2 1/4 1/8 1/16 ...

In Table 1-1 the decimal weights of the binary digit positions 0 - 15 are given. You should be able to verify the values in Table 1-1.

**Straight Binary Numbers**

A binary number is made up of a series of the binary digits 0 and 1. Examples of binary numbers are:

101
101100
110.01

The radix point is used the same as in a decimal number; it is not usually printed if no digits are to its right. The decimal value of a binary number is the sum of the all products of each digit and the decimal weight of that digit's position. This is one way of converting binary numbers to decimal:

101 = (1x4)+(0x2)+(1x1) = 5
10110 = (1x16)+(0x8)+(1x4)+(1x2)+(0x1) = 22
110.01 = (1x4)+(1x2)+(0x1)+(0x1/2)+(1x1/4) = 6 1/4

A **binary** digit is commonly called a **bit,** and a numeric suffix is added to indicate the digit position. Therefore, bit 0 is the binary digit in position zero (the units position) and bit 1 is the binary digit in position one (the twos position) and so on. There are other terms commonly used to describe the state (either 0 or 1) of a bit. When a bit is a 1, it may also be said to be **true, set,** or **on.** When a bit is a 0, it may also be said to be **false, clear, reset,** or **off.** In the following example, bit 0 is set(or on), bit 1 is reset(or clear, or off), bit 2 is set, etc. All the other bits not printed are reset(zeros).

10101

The bits of a binary number contribute a value to the magnitude of that number according to its digit position. The bit in the rightmost position is the **least significant bit,** or **LSB,** because its digit position has the smallest weight and therefore contributes least to the magnitude of the number. The bit at the leftmost position is the **most significant bit,** or **MSB,** because it contributes most to the magnitude of the number. An example is:

```
M      L
10011010
```

where the M and L respectively indicate the most and least significant bits.

The binary numbers that have been described so far are **straight binary** numbers. In straight binary all the bits are used to determine only the magnitude of the number, and none of the bits are used to indicate the sign of the number. All straight binary numbers are considered to be positive.

**Signed Binary Numbers**

Typically, a personal digital computer works with eight bits at a time, just as a calculator or adding machine may work with eight to twelve decimal digits, depending on the model. With eight bits, straight binary numbers from 00000000 ($0_{10}$) to 11111111 ($255_{10}$) can be represented. **Signed binary** is a technique to represent positive and negative numbers. Signed binary numbers indicate the sign of a number, but cannot represent as large a number in a given number of bits because the MSB is designated as the **sign bit**. When the MSB is a zero, the number is positive; when it is a one, the number is negative.

Positive signed binary numbers appear the same as straight binary numbers but with a reduced range. An eight-bit positive number can range from 00000000 ($0_{10}$) to 01111111 ($127_{10}$). Representing negative numbers is less straightforward. A negative number is formed by computing the **two's complement** of the positive number. That is, a negative number (-X) is the two's complement of the same positive number (+X).

The two's complement is calculated by first writing down the corresponding positive eight-bit binary number. Then copy the rightmost 1 bit and all the bits to the right of it, if any. Finally, write the opposite state for all the bits to the left of the 1 bit you first copied. Here are some examples of the conversion process. The top number is the positive number; the bottom is the two's complement (negative number).

$$\begin{array}{ccccccccl} &&&&&*&&& \\ 0&0&0&0&0&1&0&0&\qquad (+4_{10}) \\ >&>&>&>&>&\backslash&\backslash&\backslash& \\ 1&1&1&1&1&1&0&0&\qquad (-4_{10}) \end{array}$$

$$\begin{array}{ccccccccl} &&&&&&*&& \\ 0&0&1&1&0&0&1&0&\qquad (+50_{10}) \\ >&>&>&>&>&>&\backslash&\backslash& \\ 1&1&0&0&1&1&1&0&\qquad (-50_{10}) \end{array}$$

The * marks the rightmost 1 bit, the \ indicates a direct copying down, and the > indicates copying of the opposite state. Notice that negative numbers always have the MSB set.

The range of negative numbers that can be represented with eight bits is 11111111 ($-1_{10}$) to 10000000 ($-128_{10}$). Only 256 numbers can be represented in eight bits; this is the same for straight binary (0 - 255) and signed binary (-128 - +127). Table 1-2 shows a sequence of signed binary numbers and their decimal equivalents. As you can see, the sign bit is set in all the negative numbers, and is clear in all the positive numbers.

| Signed Binary | Decimal |
|---|---|
| 00000101 | +5 |
| 00000100 | +4 |
| 00000011 | +3 |
| 00000010 | +2 |
| 00000001 | +1 |
| 00000000 | 0 |
| 11111111 | -1 |
| 11111110 | -2 |
| 11111101 | -3 |
| 11111100 | -4 |
| 11111011 | -5 |

Table 1-2 Signed Binary and Decimal Integers.

If you are presented with a two's complement negative number and you want to know its binary or decimal value, simply compute the two's complement of the number. This will give you the positive binary value. From this you can find the decimal value by adding the products of the digits and their digit position decimal weights. For example:

| | |
|---|---|
| find decimal value of | 1 1 1 0 0 1 0 0 |
| form two's complement | 0 0 0 1 1 1 0 0 |
| bit decimal weight | 128 64 32 16 8 4 2 1 |
| decimal value = | (1x4) + (1x8) + (1x16) = 28 |
| therefore: | $11100100 = -28_{10}$ |

In preceding examples, eight bits were used to represent signed binary numbers; this can be increased to 16 or 32 or however many bits are needed to represent the range of numbers one will be working with. An eight-bit signed binary number can be represented with more bits by adding bit positions to the left and assigning them their proper state. The state of the added bit positions will be the same as the state of the original MSB, or sign bit. Here are examples of converting an eight-bit signed binary number to 16 bits:

```
                  *
8 bit             00000110      (+6₁₀)
          *
16 bit    0000000000000110      (+6₁₀)


                  *
8 bit             11111011      (-5₁₀)
          *
16 bit    1111111111111011      (-5₁₀)
```

where * indicates the sign bit. This process of repeating the sign bit in the added bit positions is called **sign extension**.

**ARITHMETIC OPERATIONS**

The arithmetic operations of addition, subtraction, multiplication, and division can be performed with binary numbers. These binary operations are quite similar to their decimal counterparts.

**Addition**

Addition is performed by adding the digits in a column, one column at a time, from right to left. If the result is less than the base, that result is put into the corresponding digit position of the sum. If the result is greater than or equal to the base, the base is subtracted from the result and the resulting digit is put into its digit position in the sum and a carry of one is added to the next column of digits to the left. This can be seen in the following decimal example:

```
 cc
 376
+ 76
----
 452
```

where 6+6 = 12 and 12-10 = 2 and a carry of one;
where a carry of 1+7+7 = 15 and 15-10 = 5 and a carry of one;
and where a carry of 1+3 = 4.

An example of binary addition, (base = 2), using this technique is:

```
     c
    1010
 +  0011
    ----
    1101
```

Beginning in the rightmost column, 0+1 = 1;
in the 2nd column, 1+1 = 2 and 2-2 = 0 with carry of one;
in the 3rd column, a carry of 1+0+0 = 1;
and in the leftmost column, 1+0 = 1.

Addition may also be performed by using a limited set of rules. There are only two digits in binary and computers typically add only a pair of numbers at one time; therefore, the number of all possible combinations of adding zeros and ones is only four. The four possible combinations of adding zeros and ones can be considered the rules of binary addition. They are:

```
 (1)      (2)      (3)      (4)
   0        0        1        1
 + 0      + 1      + 0      + 1
 ---      ---      ---      ---
   0        1        1      c 0 , or 10
```

The first three cases are straightforward; in the fourth case, one added to one is zero with a carry of one, to be added to the next column. In this case the next bit is a zero so the result is a binary 10. Examples of adding binary numbers are:

```
                c          ccc
  0100        1001          101
 +1001       + 101         + 11
 -----       -----         ----
  1101        1110         1000
```

You can double check the above examples by converting the binary numbers to decimal and seeing that the results of decimal addition are the same.

The important point in dealing with bases is to realize that binary numbering is just another way to represent a conceptual quantity. If one has five apples, the number of apples is just as validly represented with a decimal 5 or a binary 101.

If the sum of binary addition is larger than can be represented in the number of bits a computer works with, an error will result. If the sum of straight binary addition is larger than 255 (decimal) in a computer that uses eight bit positions, the correct result will not be generated. This condition is called **overflow**. An example of an eight-bit overflow as a result of addition is shown below. The values in parentheses are the decimal equivalents.

```
   c c
    1 1 0 0 0 1 1 0      (198)
+   0 1 0 1 1 0 0 1     + (89)
    -----------------    ------
    0 0 0 1 1 1 1 1   not equal to (287)
```

An overflow generated by adding straight binary numbers is detected by watching for a carry beyond the MSB. If a carry is generated from the MSB column, an overflow has occurred. If no carry is generated, there is no overflow.

With the ability to add binary numbers one can count. Table 1-3 is a list of the first 16 positive binary integers, starting with zero, and their equivalent decimal values. You should be able to reproduce this table by counting in binary; start with zero and add a 1 to generate each consecutive number.

| Binary | Decimal |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | 10 |
| 1011 | 11 |
| 1100 | 12 |
| 1101 | 13 |
| 1110 | 14 |
| 1111 | 15 |

Table 1-3 Binary and Decimal Integers.

**Another Two's Complement Technique**

We now have the ability to calculate the two's complement with a second technique. First write down a positive binary number. Then copy each bit, but in its opposite state, and add one to the result. For example:

```
8-bit positive      00000101      (+5₁₀)
                    11111010
two's             + _______1
complement =        11111011      (-5₁₀)
```

**Subtraction**

To subtract decimal numbers, you first note the sign of the number of the larger magnitude, subtract the number of the smaller magnitude from the larger, and then apply the previously remembered sign to the result. During the subtraction, you may need to subtract a larger digit from a smaller digit in a column; you do that by borrowing from the next most significant digit position. Borrowing reduces by one the digit to the left of the digit currently being subtracted from, and adds the base (ten for decimal) to the digit in the column you are currently working. An example of decimal subtraction is:

```
   b
  342
-  15
  ---
  327
```

Here 2 is less than 5, so borrow from 4 (4 - 1 = 3),
add the base to 2 (10 + 2 = 12); 12 - 5 = 7;
3 is larger than or equals 1, so 3 - 1 = 2;
and 3 is larger than or equals 0, so 3 - 0 = 3.

The b indicates a borrow. Using this same technique, an example of binary subtraction would look like:

```
   b
  1 0 1
- 0 1 1
  -----
  0 1 0
```

1 is larger than or equals 1, so 1 - 1 = 0;
0 is less than 1, so borrow from 1 (1 - 1 = 0),
add the base to 0 (2 + 0 = 2), and subtract 2 - 1 = 1;
and 0 is larger than or equals 0, so 0 - 0 = 0.

All the possible results of subracting zeros and ones in binary can be shown in four cases:

```
(1)  0    (2) b 0    (3)  1    (4)  1
    -0         -1        -0        -1
    --         --        --        --
     0          1         1         0
```

The b indicates a borrow from the next most significant position. One could use the four rules to perform binary subtraction. Some examples of binary subtraction follow. The values in parentheses are decimal.

```
                                         b
(A)      1110   (14)          (B)       1010     (10)
       - 100    (-4)                  - 0100     (-4)
        1010  (+10)                     0110     (+6)
```

```
                                      b b
(C)     00010100  (20)             00101011    (43)
      - 00101011 (-43)   =>      - 00010100   (-20)
                                  00010111    (+23)

             two's complement =  11110111     (-23)
```

In example C the result is negative, so the additional step of calculating the two's complement is required.

For a digital computer to perform these involved processes would require additional digital circuitry and increased cost. Computers that can perform straight binary subtraction may shorten the process by not determining the larger of the two numbers, generating an overflow condition if a larger number is subtracted from a smaller number and an apparently invalid result. The correct result is the two's complement of the interim result. An example:

```
b 0 1 1 0       (6)
-   1 0 1 0   - (10)
    1 1 0 0    interim result

    0 1 0 0      (-4)
```

Another subtraction technique is to add a negative number to a positive number according to the following relationship:

$$a + (-b) = a - b$$

This uses signed binary to represent positive and negative values, and follows the rules of binary addition. Examples of eight-bit subtraction using this technique are:

```
  ccc
   00100110   (+38)         00010010   (+18)
 + 11111000    (-8)       + 11101100   (-20)
   00011110   (+30)         11111110    (-2)
```

Two positive signed binary numbers can also be added to give the total in signed binary. If the result of signed binary addition is too large (either positive or negative) to fit within the number of working bits, an overflow is generated and the result is invalid. The condition of too large a negative number is also called an **underflow**. An overflow or underflow can occur only when both signed binary numbers are of the same sign and of a large enough magnitude. An overflow or underflow generated by adding signed binary numbers is detected by looking for carries beyond the two most significant bit positions. In eight bits, bits 6 and 7 are the two MSBs. If both bit positions either do or do not generate a carry, there is no overflow or underflow. If one bit position generates a carry and the other does not, there is an overflow or underflow. Below are two examples of adding signed binary numbers. Example A generates an overflow condition and B generates an underflow.

```
(A)    cc ccc          (B)  c
      01010011               10010110
    + 00110101             + 11001000
      --------               --------
      10001000               01011110
```

**Multiplication**

The process of straight binary multiplication is very similar to decimal multiplication. The main difference is in the binary multiplication table, which follows:

```
   0       0       1       1
 x 0     x 1     x 0     x 1
 ---     ---     ---     ---
   0       0       0       1
```

Multiplication is performed by multiplying each bit of the multiplicand by each bit of the multiplier using the binary multiplication table. The result is placed underneath each multiplier bit position and extended to the left. Then the results of each bit multiplication are added together to give the final answer. Examples of this process are:

```
(A)                              (B)
   00101101   (45)                    00010100    (20)
  x       11   (3)                   x     101     (5)
  ----------                         ---------
   00101101                           00010100
  00101101                           00000000
  ---------                         00010100
  010000111   (135)                 ----------
                                    0001100100    (100)
```

When multiplying, be careful that the result fits within the number of bits a computer uses. The largest result (in straight binary) that can be represented is 11111111 ($255_{10}$) in eight bits and 1111111111111111 ($65535_{10}$) in 16 bits. If the result is larger than the limit, the most significant one bits that fall beyond the number of working bit positions will be lost, and the final result will be meaningless.

When the multiplier has only one bit set in it, multiplying shifts the multiplicand to the left N bit positions with respect to the radix or **binary point** (where N is the bit position of the set bit in the multiplier). Examples of binary numbers with only one set bit are 10, 100, and 1000. Put another way, multiplying by 2, 4, or 8 is done by shifting the binary point of the multiplicand to the right one, two, or three positions. Multplication of signed binary numbers is not often attempted because of the difficulties of handling the sign bit.

**Division**

Division is the process of determining how many times one number (the divisor) is contained in another number (the dividend). Binary division is performed much like decimal long division, in that one uses a trial and error method to see how many times the divisor is contained in the dividend. In binary long division, the divisor either is (1), or is not (0) contained in a group of bits. Consider the following example:

```
1010 / 10  =      0101
             10 / 1010
                  -0
                   10
                  -10
                    01
                    -0
                     10
                    -10
                      0
```

The division process follows: 10 is not contained in 1 (the MSB of the dividend), so put a 0 (MSB) in the quotient area, subtract 0 x 10 from 1, and bring down the next dividend bit (0). 10 is contained in 10, so put a 1 in the quotient area, subtract 1 x 10 from 10, and bring down the next dividend bit (1). 10 is not contained in 1, so put a 0 in the quotient area, subtract 0 x 10 from 1, and bring down the next dividend bit (0). 10 is contained in 10, so put a 1 in the quotient area, subtract 1 x 10 from 10 leaving a 0 indicating that 10 evenly divides into 1010.

If the divisor has only one bit set in it, as in the last example, the quotient is the dividend shifted to the right N bit positions with resptect to the binary point ( N is the bit position of the bit set in the divisor).

Put another way, division can be accomplished by shifting the binary point of the dividend to the left if the divisor is binary 10, 100, 1000, etc., or decimal 2, 4, 8, 18, etc.

## LOGICAL OPERATIONS

We have seen how bits can represent numbers; as you remember, bits were originally used to represent the state of electronic switches. Just as bits are used to represent the computer's internal switch states, we could, by analogy, use a bit to indicate the state or condition of anything we choose. The state of a bit is determined by asking if an event or situation is true or false. If the statement, it is raining, is true, the bit is set to 1. If it is false, the bit is cleared to 0. A number of bits could be used to represent a number of events as follows:

```
bit 0 indicates it is raining
bit 1 indicates it is cloudy     = 1001
bit 2 indicates I am indoors
bit 3 indicates I am hungry
```

The four bit code above, read from left to right, reveals that: I am hungry, I am not indoors, it is not cloudy, and it is raining. Depending on the conditions represented, the bits are set or cleared with no regard for the state of any other bit. This is quite unlike any arithmetic operations we have covered; these nonarithmetic operations are called **logical operations.**

### Complement

The logical operation of taking a complement is to reverse the bit states. Forming the complement of a bit or a series of bits is performed by replacing each 1 with a 0, and each 0 with a 1. The symbol that indicates a complement is to be taken is a bar over the quantity to be complemented:

$$\overline{1} = 0$$

$$\overline{0110} = 1001$$

Since a bit can be in only one of two states, a complemented bit is not in its original state. This may seem obvious, but the concept is useful in representing logical relationships:

$$\overline{X} = \text{not } X$$

$$\overline{\text{it is raining}} = \text{it is not raining}$$

$$\overline{\text{true}} = \text{false}$$

X is a condition represented by a bit. The complement of an event may also be called **not** an event.

**Inclusive Or**

Inclusive or (**OR**) is a logical operation performed between two or more bits, although digital computers usually perform an inclusive or between only two bits. The four possible cases of inclusive oring a pair of bits are:

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

where OR indicates that the inclusive or operation is to be performed between the two bits surrounding it. As you can see, the result of inclusive oring two bits is 1 if any of the original bits are 1. The inclusive or is also known simply as an OR operation. A truth table can also be used to represent this logical operation on two bits, A and B, as seen in Fig. 1-1.

| B \ A | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

Fig. 1-1 The OR Operator Truth Table.

The top row of Fig. 1-1 shows the possible states of A, and the leftmost column shows the possible states of B. The value of A OR B can be found at the intersection of the column of the particular value of A and the row of the particular value of B. In a computer that works with eight bits, a group of eight bits is ORed with another group of eight bits. The ORing process is done between bits of the same bit position as follows:

$$\begin{array}{l} 01101001 = X \\ \underline{00110001} = Y \\ 01111001 = X \text{ OR } Y \end{array}$$

**And**

The AND operation between a number of bits results in a 1 only if all the bits being ANDed are 1. The four possible cases of ANDing two bits are:

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

The truth table for the AND operation of bits A and B is shown in Fig. 1-2 where you can see that A AND B is true only if A and B are both true.

| B \ A | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

Fig. 1-2 The AND Operator Truth Table.

The ANDing of two groups of bits is done by ANDing pairs of bits in the same bit position. An example is:

$$\begin{array}{l} 01101001 = X \\ \underline{11010011} = Y \\ 01000001 = X \text{ AND } Y \end{array}$$

**Exclusive Or**

The exclusive or (**XOR**) operation performed on two bits results in 1 only if just one of the bits is a 1. The four possible cases are:

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

The truth table for the XOR operation of bits A and B is shown in Fig. 1-3 where you can see that A XOR B is true only if only A or only B is true.

| B \ A | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

Fig. 1-3 The XOR Operator Truth Table.

Exclusive oring two eight-bit groups is performed between pairs of bits in the same bit position, as seen below:

$$\begin{array}{l} 01101001 = X \\ \underline{01011101} = Y \\ 00110100 = X \text{ XOR } Y \end{array}$$

**Boolean Algebra**

The set of rules governing the use of logical operations is called

Boolean algebra. The logical operators AND, OR, and NOT and the numbers 0 and 1 are used in Boolean algebra. The operators AND and OR are represented with the symbols, $\wedge$ and $\vee$, respectively. Sometimes the AND symbol, $\wedge$, is not used at all; instead the two quantities are just written next to each other. The new symbols and their usage are shown:

$$\begin{array}{c} A \text{ AND } B = A \wedge B = AB \\ A \text{ OR } B = A \vee B \end{array}$$

The exclusive OR operation is indicated with the $\veebar$ symbol.

The basic rules of Boolean algebra using binary numbers are:

$$\begin{array}{llll}
1) & \overline{0} = 1 & 2) & \overline{1} = 0 \\
3) & A \wedge 1 = A & 4) & A \vee 0 = A \\
5) & A \wedge A = A & 6) & A \vee A = A \\
7) & A \wedge 0 = 0 & 8) & A \vee 1 = 1 \\
9) & A \wedge \overline{A} = 0 & 10) & A \vee \overline{A} = 1 \\
11) & \overline{\overline{A}} = A & &
\end{array}$$

A is a Boolean variable that can represent a bit; the double bar over the A in rule 11 indicates that A has been complemented twice. Notice the pairs of rules: 1 and 2, 3 and 4, 5 and 6, 7 and 8, and 9 and 10. Given half of a pair, the other half can be found by exchanging 0 and 1, and the operators, $\vee$ and $\wedge$. For example:

$$\begin{array}{ll}
7) & A \wedge 0 = 0 \\
 & \quad \backslash \, \backslash \quad \backslash \\
8) & A \vee 1 = 1
\end{array}$$

The \ indicates copying down the opposite symbol. This is the principle of **duality**, or rule 8 is the dual of rule 7.

The Boolean operations of $\wedge$ and $\vee$ follow the laws of **commutation**, **association**, and **distribution**. The commutative law states that the result of an algebraic operation is independent of the order in which it is performed. This is expressed as:

$$\begin{array}{c} A \vee B = B \vee A \\ A \wedge B = B \wedge A \end{array}$$

The associative law states that the result of a series of identical operations is independent of the order in which it is performed:

$$A \vee (B \vee C) = (A \vee B) \vee C$$
$$A \wedge (BC) = (AB) \wedge C$$

The operations within parentheses are performed first.

The distributative law states that the result of an algebraic operation between a Boolean variable A and a group of variables equals the result of the algebraic operation between the variable A and each variable within the group. Below are two examples of this law:

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$
$$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$$

By inspecting the above examples, you can see that each pair consists of a Boolean expression and its dual. Here are the rest of the Boolean laws or equalities:

12) $A \vee \overline{A}B = A \vee B$  13) $A \wedge (\overline{A} \vee B) = A \wedge B$

14) $A \veebar B = \overline{A}B \vee A\overline{B}$  15) $\overline{A \vee B} = \overline{A} \wedge \overline{B}$

16) $\overline{A \wedge B} = \overline{A} \vee \overline{B}$

Rule 13 is the dual of 12, and 16 is the dual of 15. Rules 15 and 16 are known as **De Morgan's** law.

The rules and laws of Boolean algebra will be useful to help you simplify complicated expressions, thereby reducing the amount of work you and the computer must do. Here's an example: you are in charge of putting a toddler to bed. You have determined that the following facts and steps will guide you in this chore.

A = the toddler is tired
B = it is the toddler's bedtime
C = put the toddler to bed

After giving it careful thought, you have decided that you will put the child to bed (C is true) under any of the following conditions: when he is tired and it is not his bedtime, or when he is not tired and it is his bedtime, or when he is tired and it is his bedtime. This can be represented as:

$$C = (A \wedge \overline{B}) \vee (\overline{A} \wedge B) \vee (A \wedge B)$$

The expression can be simplified by using the laws of Boolean algebra.

Using the associative law we arrive at:

$$C = A(\overline{B} \vee B) \vee \overline{A}B$$

Using rules 10 and 3 the above equation reduces to:

$$C = A \vee \overline{A}B$$

Then, applying rule 12 gives us the final equation:

$$C = A \vee B$$

We now see that the toddler should be put to bed if he is tired or if it is his bed time. Had you programmed a computer to make this decision for you, it would have been much more difficult for you to direct it to solve the original equation, and the computer would have taken longer to find the solution.

## HEXADECIMAL NUMBERS

Most digital computers work with groups of eight bits, or multiples of eight, i.e., 16, 24, 32, bits. A group of bits, inside the computer, is composed of a group of electronic switches whereby the group as a whole can be controlled or manipulated. A group of eight bits is known as a **byte**. Generally, any group of bits controlled as a whole is called a **word**. The word **length** is how many bits are in that group. So, a byte is an eight-bit word. A 16-bit group is called a double byte, or a 16-bit word or a word sixteen bits long. The Color Computer uses bytes and double bytes. A byte and double byte are graphically shown in Fig. 1-4.

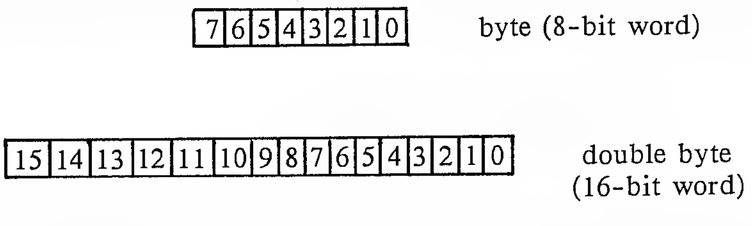


Fig. 1-4 A byte and a double byte.

In Fig. 1-4 each box is numbered according to its bit position. A byte can also be divided into two halves, and each half is called a **nibble**. The nibble composed of bits 7 - 4 is the upper or most significant, and the nibble composed of bits 3 -0 is the lower or least significant. A nibble may also be considered to be a four-bit word. A 16-bit word is divided into four nibbles, where the least significant is nibble 0 and the most significant is nibble 3. Figure 1-5 illustrates how bytes and double bytes are divided into nibbles.

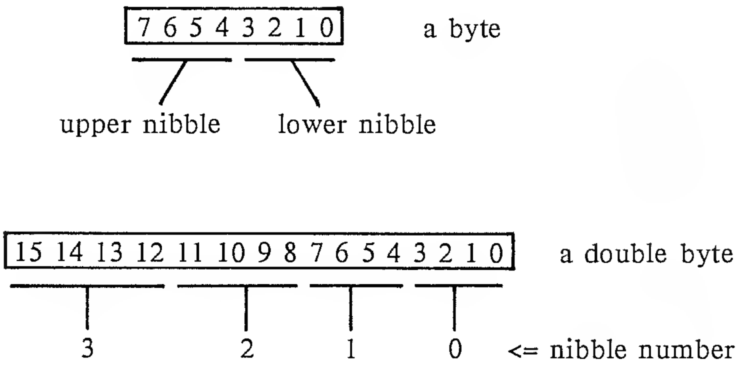


Fig. 1-5 The locations of nibbles.

The decimal quantity of numbers that can be represented with a group of bits is the decimal number of unique combinations of zeros and ones that can be constructed within that group of bits. For a group of N bits there are $2^N$ unique combinations. That means that a nibble can represent one of 16 ($2^4$) decimal numbers, a byte can represent one of 256 ($2^8$) decimal numbers, and a double byte can represent one of 65,536 ($2^{16}$) decimal numbers. Also, if a group of bits is expanded to include one more bit, the quantity of numbers it can represent is doubled.

So far we have been writing in binary to represent binary numbers, but if we were to write very large numbers or a lot of smaller numbers, the strings of zeroes and ones would become quite long and cumbersome. There is a shorthand method to represent binary numbers, but it entails the use of another numbering system. This is the **hexadecimal** number system, whose base is 16 and is composed of 16 digits. Table 1-4 lists the 16 hexadecimal digits and their equivalent binary and decimal values. You should memorize this table.

## Converting Binary To Hexadecimal

To convert a binary number to a hexadecimal number, let each nibble be represented by its hexadecimal equivalent digit. Some examples are:

0 0 1 0 1 0 1 0 binary byte
2 A = $2A_{16}$ hexadecimal

0 0 1 1 1 1 1 1 0 1 1 0 1 1 0 0 binary double byte
3 F 6 C = $3F6C_{16}$ hexadecimal

Notice that the hexadecimal representation is much more compact.

| Binary | Decimal | Hexadecimal |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |
| 1000 | 8 | 8 |
| 1001 | 9 | 9 |
| 1010 | 10 | A |
| 1011 | 11 | B |
| 1100 | 12 | C |
| 1101 | 13 | D |
| 1110 | 14 | E |
| 1111 | 15 | F |

Table 1-4 Hexadecimal, Binary, and Decimal Numbers.

**Converting Hexadecimal To Binary**

To convert hexadecimal to binary, just reverse the above procedure. Replace each hexadecimal digit with its equivalent four-bit binary code from Table 1-4. Examples of this are:

```
   2    A     hexadecimal
0010 1010     binary byte

   D    A    1    7      hexadecimal
1101 1010 0001 0111      binary double byte
```

**Converting Hexadecimal To Decimal**

To convert hexadecimal to decimal, one could first convert to binary, as above, and then convert from binary to decimal. One could also convert directly from hexadecimal to decimal by summing the products of each digit's decimal value and the decimal weight of its digit position. The decimal value of each digit can be determined from Table 1-4. The decimal weight of each digit position is 16 raised to the power of the digit position. The digit positions are: . . . 3 2 1 0. The decimal weight of each digit position is: . . . $16^3$ $16^2$ $16^1$ $16^0$, or: . . . 4096 256 16 1. Examples of converting from hexadecimal to decimal are:

$7C = (7 \times 16) + (12 \times 1) = 124_{10}$
$F2 = (15 \times 16) + (2 \times 1) = 242_{10}$
$31A2 = (3 \times 4096) + (1 \times 256) + (10 \times 16) + (2 \times 1) = 12706_{10}$

**Changing The Base**

A generalized technique exists for converting a number in any base to a number in any other base. A number N in base x may be converted to base y by a series of divisions of N by y, with the division performed in base x. The remainders from the divisions will be the digits of the number in base y.

```
y \_N__
  y \_N₁        A₀
    y \_N₂      A₁
      y \_N₃    A₂
          .
          .
```

The remainders $A_i$, which must be smaller than y, are the digits of the converted number as follows: . . . $A_2 \ A_1 \ A_0$. As an example, let us convert $688_{10}$ to hexadecimal.

```
16 \_688       remainders
  16 \_43        0
    16 \_2       11 = B (from Table 1-4)
          0      2
```

therefore $688_{10} = 2B0_{16}$

In the above example, 688 divided by 16 is 43, with a remainder of 0, 43 divided by 16 is 2, with a remainder of 11, and 2 divided by 16 is 0, with a remainder of 2.

**Arithmetic Operations**

The normal arithmetic operations of addition, subtraction, multiplication, and division can be done with hexadecimal numbers, though the operations are more cumbersome because the base is 16. The same technique of adding numbers is used. When performing hexidecimal addition, a carry is generated when the sum of a column of digits is equal to or greater than 16, decimal. The total of the column generating the carry is then reduced by 16, the base. Some examples are:

```
   c            c
  2 F        1 A 8 4
+ 0 1      + 2 3 E 2
  3 0        3 E 6 6
```

The subtraction process is performed like that described for decimal and binary numbers. When a borrow is generated, remember to add 16 to the digit being subtracted from in the column that generated the borrow. Also, decrement the digit being borrowed from. Two examples are:

$$\begin{array}{rcc} & \mathrm{b} & \\ & 3 & 3 \\ - & 0 & 5 \\ \hline & 2 & \mathrm{E} \end{array} \qquad \begin{array}{rcccc} & & \mathrm{b} & & \\ & 2 & \mathrm{A} & 5 & \mathrm{C} \\ - & 1 & 8 & \mathrm{E} & 7 \\ \hline & 1 & 1 & 7 & 5 \end{array}$$

You can check the results of hexadecimal arithmetic by converting the examples to binary and performing the operations again.

# CHAPTER 2

# Memory and Data Representation

We have learned how to manipulate bytes of data; but where do the bytes come from, and where are the results of manipulation stored?

## MEMORY

Bytes of data for immediate or near term use are stored in the **random access memory**, or **RAM**. The memory is called random access because any byte stored in it is just as accessible as any other byte. The memory is a major subunit of a digital computer, and is usually made up of thousands of electronic circuits. Because electronic circuits don't work without electrical power, all bytes stored in memory are lost when power is turned off, even if only momentarily. A memory in which data storage depends on the presence of power is said to be **volatile**. Fig. 2-1 is a simplified block diagram of a microcomputer, depicting two major subunits, the **microprocessing unit**, or **MPU**, and the random access memory. The MPU is the section of the computer that performs the arithmetic and logical operations on the bytes of data.

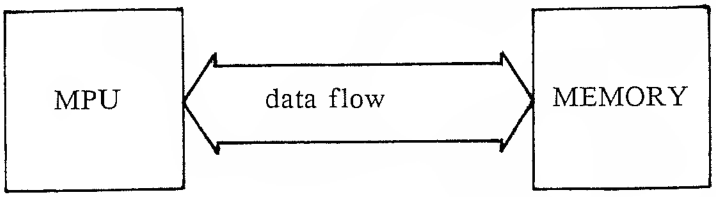


Fig. 2-1 A Simplified Microcomputer Block Diagram.

The two operations the memory is capable of are **storing** and **reading**. The store, or **write**, operation is the process of putting a byte into memory where it will be retained until it is replaced. The read, or **fetch**, operation is the process of extracting a byte from memory. A typical sequence of operations would be: the MPU reads a byte from memory, performs an arithmetic operation on that byte, and then stores the resultant byte in memory.

We have seen that bytes can be stored in and read from memory, but we need also to know where in memory the byte will be stored or read. Let's learn how the memory is organized: conceive of the memory as composed of **cells**, in each of which one and only one byte can be stored. To differentiate one cell from another, all the cells are numbered from 0 to N-1 where the memory contains N cells. Fig. 2-2 depicts a memory of 16 cells where the cells are numbered from 0 to F, hexadecimal.

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | A | B |
| C | D | E | F |

Fig. 2-2 A Memory with Hexadecimal Addresses.

The memory in Fig. 2-2 is drawn for convenience as a square with four cells on a side. When performing a read or write operation one must specify the number of the cell the computer should access. This number of each cell is called its **address** or **location**. For example, one could store a byte in address one (cell number one), or store a double byte in addresses four and five, where the most significant byte would be put in address four and the least significant byte in address five. An example of memory usage would be to read a byte from address $5_{16}$, manipulate that byte, and storing the resulting byte in address $B_{16}$.

Whatever byte is stored at an address is currently the **contents** of that address. When a byte is read from an address, the content is not changed. This is analogous to playing a cassette tape; after listening to a cassette tape, whatever was recorded on the tape has not been changed. Writing puts a new byte in the specified cell and destroys its previous contents.

Let's turn on the Color Computer and experiment with writing to and reading from memory. The BASIC command for writing into memory is POKE X,Y where X is the decimal number or expression of the address and Y is the decimal value or expression of the byte to be stored. Type the following command:

```
POKE 3000,185  (ENTER)
```

(ENTER) indicates that the ENTER key must be depressed. All the following type-in examples require the ENTER key to be depressed.

You have just stored a straight binary byte of decimal value 185 in decimal address 3000. The BASIC function for reading the content of a memory address is PEEK (Z), where Z is the decimal number or expression of the address to be read. Let's read and print the location we have just written to. Type the following command:

```
PRINT PEEK(3000)
```

You will see the value of the byte in address 3000 printed on the screen, that is, the 185 we previously stored there. BASIC is designed to work with decimal numbers, but we can indicate that a number is hexadecimal by preceding it with &H. The hexadecimal number equivalent to decimal 3000 is BB8. You can check this with one of the conversion techniques covered in Chapter 1. We can read and print the content of the original address by typing:

```
PRINT PEEK(&HBB8)
```

You will see 185 printed, the value we originally stored in the address. To have the values printed in hexadecimal, the variable to be printed must be modified with the HEX$ function, as below:

```
PRINT HEX$(PEEK(&HBB8))
```

This will print B9 as the hexadecimal value of the byte at address BB8. Convert B9 to decimal and you will see it is equal to 185. A more compact way of indicating a hexadecimal number in text is to prefix the number with a dollar sign ($). Let's store $C2 at address $1000:

```
POKE &H1000,&HC2
```

To read and print the contents of address $1000, type:

```
PRINT HEX$(PEEK(&H1000))
```

This prints C2, the hexadecimal value of the byte we stored at address $1000. The PRINT HEX$ command has a fault, demonstrated as follows:

```
POKE &H1000,&H0F
```

This stores a byte of $0F at $1000. Now print the content by typing:

```
PRINT HEX$(PEEK(&H1000))
```

You will see F is printed instead of 0F. The PRINT HEX$ command does not print leading zeros.

Experiment more by yourself, but be sure to use addresses larger than $600, because BASIC itself uses the addresses below this. If you try to store a decimal value larger than 255 or less than zero, you will get an FC error message because the value is outside the range that can be represented in one byte in straight binary.

**More About Memories**

When someone asks how much memory your computer has, he is actually asking how many cells or addresses its memory has. Your response could be the number of cells in decimal, such as 16,384. You might also answer 16K, where one K is 1024 decimal. Notice that 1024 is the decimal weight of the bit position ten. So 16K is:

$$16K = 16 \times 1024 = 16384$$

The maximun amount of memory the Color Computer can work with is 64K, or 65,536 memory locations. These addresses are numbered from 0 - 65,535 in decimal or from 0000 - FFFF in hexadecimal. Notice that it takes exactly two bytes to represent the address.

**Read Only Memory**

Another type of memory is known as read only memory or **ROM**. One can not alter the contents of read only memory. A ROM has specific data stored, at the time of its manufacture, in each of its locations that can not be changed; one can only read from this type of memory. In the Color Computer the internal ROM occupies addresses $8000 - $BFFF. This ROM contains Color BASIC and Extended Color BASIC, the programs that interpret and execute your BASIC program's statements. Verify that this is indeed read only memory by reading the content of an address in ROM and trying to store another value in its place. A demonstration using BASIC PEEK and POKE commands is:

```
PRINT PEEK(&H9000)
132                    (observe original value)
POKE &H9000,92               (try to change it)
PRINT PEEK(&H9000)
132                    (observe it was not changed);
```

Addresses above $BFFF are reserved for use by plug-in ROM packs such as games, EDTASM+, or a disk controller.

A diagram depicting the memory addresses, their use, and memory type can be seen in Fig. 2-3. This diagram is known as a **memory map**; it shows how memory is organized and used. Memory locations \$0000 - \$7FFF of a 32K (\$0000 - \$4FFF for 16K) Color Computer are RAM. As we learn more we will be able to add detail to this map by indicating what some of the addresses are used for.

| Decimal Address | Hexadecimal Address | Memory Type |
|---|---|---|
| 0 | 0000 | RAM |
| 16383 | 4FFF | |
| | | RAM |
| 32767 | 7FFF | |
| 32768 | 8000 | Internal ROM |
| 49151 | BFFF | |
| 49152 | C000 | External ROM |
| 65535 | FFFF | |

Fig. 2-3 A Memory Map of the Color Computer.

## DATA REPRESENTATION

We can represent a number with a byte or a double byte using straight binary or signed binary. A computer memory is capable of storing thousands of bytes representing different types of data. Many conventions exist concerning how data should be represented and stored in memory. Each convention has strong points and weak points, making it useful or not for a particular situation.

**Straight Binary**

Straight binary lets us represent only positive integers. As you know, one byte in straight binary can represent decimal integers from 0 - 255. To represent larger numbers, more bytes must be used. For instance, two bytes can represent decimal integers from 0 - 65535. Suppose we had measured the daily wind speed, in integer miles per hour, and wanted to store the speed in memory. One byte in straight binary would represent the wind speed for one day, since the speeds would be within the permissible range and would never be negative. The data bytes would be stored in sequential memory locations - that is the first day's wind speed would be stored at address 1000, the second day's speed at address 1001, the third day's speed at address 1002, and so on. Sequentially stored bytes are typically depicted as below:

```
1000    07  12  03  05
1004    12  17  08  06
```

Each row is preceded by the address in which the first byte is stored. The second byte is stored in the next higher address, and each succeeding byte is stored in the next sequentially higher address. In our example, the first day's wind speed is 7 miles per hour, stored at address 1000; the second day's speed is 12, stored at address 1001, the third day's speed is 3, stored at address 1002, and so on. There is no rule saying how many addresses would be in each row; normally, the length of a row would be such that it made sense or was easy to read. In fact, the above example should probably have had seven addresses in each row so each row would correspond to a week.

If a number is so large that two bytes are required to represent it, the two bytes would be stored in two consecutive addresses, the most significant byte at address N and the least significant byte at address N+1. If three bytes were required they would be stored in three consecutive addresses. A two byte value ($1C27) and a three byte value ($023A1B) are shown stored in memory below:

```
2000    1C  27
2002    02  3A  1B
```

The straight binary technique uses the least amount of memory to store numbers, because all the bits are used to determine the value of a number, and none are used to indicate the sign or where the radix point is. Also, a program that uses straight binary values will execute, or run, faster because the quantities are in the form the MPU most readily works with and the least number of memory addresses have to be read.

**Signed Binary**

Signed binary has an advantage over straight binary because it can represent negative integers. One byte can represent any decimal integer from -128 - +127. Two bytes can represent any decimal integer from -32768 - +32767. A signed binary byte can be stored anywhere in memory, or a series of bytes can be stored in consecutive memory locations. A number represented by two or more bytes is stored in memory by putting the most significant byte at address N and the next most significant bytes at addresses N+1, N+2, etc. This can be shown as:

```
1000    FF  FC
1002    00  12
```

Decimal -4 is represented by two bytes stored at addresses 1000 and 1001, and a decimal +18 is represented by two bytes stored at addresses 1002 and 1003.

Remember that the most significant bit is the sign bit; if it is set the number is negative, and if it is clear the number is positive. A number represented by one byte can be represented by two or more bytes by using the process of sign extension. This is done by repeating the sign bit in all the bits of the new bytes prefixed to the original byte. Two examples are:

| | | | | |
|---|---|---|---|---|
| A3 | $(-93_{10})$ | => | FFA3 | $(-93_{10})$ |
| 6C | $(+108_{10})$ | => | 006C | $(+108_{10})$ |

The bytes are shown in hexadecimal.

As in straight binary, signed binary lets you pack nearly the most data into the least amount of memory. An added plus is the ability to represent negative numbers.

## Binary Coded Decimal

Sometimes it is desirable to use and manipulate decimal numbers in the computer. This can be done using **binary coded decimal**, or **BCD**, where a decimal digit is represented by the four bits in a half byte or nibble. The first ten hexadecimal numbers, 0 - 9, are considered to be decimal digits and the other numbers, A - F, are not allowed. In BCD a byte can represent two decimal digits, or N bytes can represent 2xN decimal digits. Some examples of BCD are:

| Byte | => | BCD Value |
|---|---|---|
| 73 | | 73 |
| 12 | | 12 |
| 0389 | | 389 |
| 1C77 | | not a valid BCD number |

The bytes are stored in memory as before, the most significant byte at address N and the next most significant bytes at addresses N+1, N+2, etc:

```
1020   02 19
1030   00 92 01
```

A double byte representing decimal 219 is stored at 1020 and 1021, and a triple byte representing decimal 9201 is stored at 1030, 1031, and 1032. The technique of using each nibble to portray a decimal digit is also called **packed** BCD. As you can see, only positive whole numbers can be represented with BCD as we know it so far.

The arithmetic operations of addition and subtraction can be performed on BCD numbers. For addition, a microcomputer simply adds the two numbers together as though they were straight binary.

```
   7 2        0 1 0 2
 + 2 5      + 3 7 4 4
   9 7        3 8 4 6
```

Sometimes adding two BCD digits will result in an illegal digit, that is, one of the six unused hexadecimal digits of A - F. To correct this, six is added to the illegal digit, and any generated carry is added to the next digit to the left. Two examples follow:

```
   2 8        1 0 7 3
 + 1 7      + 2 0 5 5
   3 F        3 0 C 8
 +   6      +   6
   4 5        3 1 2 8
```

You can see that the final results are the sums of the decimal numbers.

A microcomputer will perform BCD subtraction by doing a straight binary subtract or we can do it on paper using hexadecimal subtraction. Again, the result of subtraction may generate an illegal BCD digit. This digit is corrected by subtracting six from it. This correcting process can be below.

```
   b          b
   7 3        2 3 7 6
 - 0 5      - 1 7 5 5
   6 E        0 C 2 1
 -   6      -   6
   6 8        0 6 2 1
```

The process of correcting illegal BCD digits after adding or subtracting is known as **decimal adjust**.

Often the number of digits making up a BCD number may not be known. Thus, it is not known how many bytes to read from memory to assemble the complete number. It would also be helpful to be able to assign a sign to a number, and to have a decimal point. These items can be included in a BCD number at the expense of using more memory cells. This can be done as seen below:

```
SS PP DD DD DD DD EE
```

where the first byte, on the left, (SS) would indicate the sign (say a 00 for positive and a 01 for negative) and the second byte (PP) would contain a count of decimal digits to skip, going left to right, to place the decimal point. These two bytes are followed by the BCD digits (D) and an end code (EE). An example of this as stored in memory is:

$$2000 \quad 01\ 05\ 29\ 21\ 44\ 30\ EE = -29214.43_{10}$$

The advantage of this technique, called **expanded BCD**, is the ability to store an exact copy of any decimal number in memory. Two disadvantages are that BCD uses a lot of memory to store a number, and that programs using BCD numbers execute more slowly because of the larger number of bytes that must be read and manipulated.

**Fixed Point Representation**

The numeric representation techniques of straight binary, signed binary, and packed BCD are only capable of representing integers, except that signed binary can also indicate the sign. You have probably considered a group of integers to be a whole number, even though no decimal or radix point was indicated. You assumed the radix point was understood to be at the far right of the group of integers.

One could, however, fix the radix point at any desired position in a field of digits. After storing the data as a string of integers in memory, and later retrieving it, the radix point could be reinserted in its position. Suppose you wanted to store the daily rainfall amount in memory using packed BCD. You could construct a field of digits such as XX.YY, where XX is inches of rainfall and YY is hundredths of inches of rainfall. The integer group XXYY would then be stored in memory. On retrieving this group of integers the decimal point would be reinserted to give XX.YY. This technique of fixing and remembering the position of the radix point is known as **fixed point** representation. The arithmetic operations performed with straight binary, signed binary, and BCD are also known as fixed point arithmetic.

The advantage of fixed point operations is that results can be exactly correct, assuming enough bytes are used so an overflow condition does not occur. A disadvantage exists when one uses fixed point representation to portray a numeric variable with a large or very large range, for example, the count of bacteria in a quart of water. This count could range from zero to several million. To store this variable in memory using BCD, one would have to designate a field of four bytes to hold eight decimal digits. Some typical bacteria counts stored in memory might look like this:

```
2000   00 01 79 20
2004   01 10 27 36
2008   00 00 01 42
```

As you can see, the bacteria count of 1,102,736 at address 2004 uses all the available bytes, but the lower valued counts do not use all the reserved bytes. The bytes containing leading zeros or trailing zeros are wasted.

**Floating Point Representation**

Often a number or a result of a calculation need not be exactly correct. The quantities need be accurate only to so many decimal places, depending on the amount of accuracy required. For instance, calculations involved in building a bridge may not need to be any more accurate than a tenth of an inch, because the wind or ground settling will cause distortions of greater than a tenth of an inch. The technique used by digital computers to represent numbers to a certain accuracy is called **binary floating point**. The advantage of binary floating point is it can represent a large range of numbers with a fixed number of bytes.

First let's review decimal floating point, also known as **exponential** or **scientific notation**. A floating point number is written in the form:

$$+/- \text{N.NNN...} \times 10^{+/-E}$$

where its value is equal to the number N.NNN... multiplied times 10 raised to the power of E. The number N.NNN... is the **mantissa** and the exponent, E, is the **characteristic**, or simply, the exponent. Any decimal number that can be represented as a string of digits and an appropriately placed decimal point can also be represented with decimal floating point. Some examples of fixed point decimal numbers and their floating point equivalents are:

$$+ 12 = 1.2 \times 10^1$$
$$- 542.7 = - 5.427 \times 10^2$$
$$+ .000678 = + 6.78 \times 10^{-4}$$

In the first example we can see that 12 is indeed equal to 1.2 times 10 ($10^1$). In the examples the mantissas have been **normalized** or adjusted so their value is greater than or equal to one and less than ten. The mantissa is normalized according to convention. It is the job of the exponent to indicate the general magnitude of the number. The mantissa determines the accuracy, dependent on the number of decimal places in it. The sign of decimal floating point is simply indicated with a plus or minus sign to the left of the mantissa.

Binary floating point is written in the form:

$$+/- \ .NNN... * 2^{+/-E}$$

where the mantissa .NNN... is a binary number normalized so the bit to the right of the radix, or **binary point** is always 1. The value is found by multiplying the mantissa by 2 raised to the power of E. Some examples of finding the binary value of binary floating point numbers are:

$$.110x2^3 = .110x8_{10} = .110x1000_2 = 110.$$
$$.1011x2^{-2} = .1011x1/4_{10} = .1011x.01_2 = .001011$$
$$.1001x2^5 = .1001x32_{10} = .1001x100000_2 = 10010.$$

A fixed point binary value can also be found by moving the binary point of the mantissa to the right if the exponent is positive, or to the left if the exponent is negative. The number of bit positions to move it is the magnitude of the exponent. What was the mantissa will be the binary value. Try the above examples using this technique.

Shifting the binary point can also be used in reverse to convert a fixed point binary value to floating point. Given a fixed point binary number, shift the binary point so the result is normalized; that is, the leftmost 1 bit is just to the right of the binary point. The number now is the mantissa. The exponent is the number of bit positions the binary point was shifted. The sign of the exponent is plus if the binary point was shifted to the left, or minus if to the right. An example of this conversion process is:

1) Given 1010.
   Mantissa = .1010 (binary pointt
                  shifted left 4 positions)
   Exponent = 4 (number of shifts)
   Sign of exponent = + (left shift)
 Therefore, 1010. = .1010 x $2^{+4}$

Now we need to be able to store floating point in memory. One way would be to reserve three bytes for each number. The most significant byte would contain the exponent(EE) in signed binary to accommodate positive or negative exponents. The next two bytes could contain the normalized mantissa (NNNN) in straight binary where the binary point is always at the far left. For example, this could be stored in memory at addresses 1000, 1001, and 1002 as:

```
1000   EE NN NN
```

A specific number could be stored in memory as:

The floating point number = .11011 x $2^{-7}$
The mantissa .11011 = D8 00
The exponent = F9 (two's complement of decimal 7)
So, at 1000: F9 D8 00

The limitation of this example is that only positive binary floating point numbers can be represented.

The degree of accuracy of a floating point number depends on the number of digit positions in the mantissa. In the last example there are two bytes, meaning that the accuracy is limited to about one in 65536 or .0015 percent. This is because the mantissa is composed of sixteen bits, within which a decimal number from 0 - 65535 can be represented. The accuracy can be increased or decreased by respectively increasing or decreasing the number of bytes reserved for the mantissa. The range of the magnitude is determined by the possible range of the exponent. In the example the exponent could range from -128 - +127. This is approximately equivalent to a decimal range of $10^{38}$ to $10^{-38}$. If more or fewer bits were reserved for the exponent, the range would be more or less.

**Character Representation**

The alphabetical characters are also stored, manipulated, transmitted, and printed by computers. These characters include the alphabet - upper- and lowercase forms; punctuation marks; numeric symbols; and special characters such as #, <, and ]. Two standardized codes for internally representing characters have been adopted. One is **EBCDIC**, the Extended Binary Coded Decimal Interchange Code, which is primarily used in larger computers. EBCDIC is an eight-bit, or one byte, code that allows the representation of up to 256 different characters. The other code is **ASCII**, the American Standard Code for Information Interchange, the code used by most microcomputers. ASCII is a seven-bit code by which up to 127 different characters can be represented. The seven-bit code is implemented using the seven less significant bits in a byte as seen below:

$$\boxed{7\ \underbrace{6\ \ 5\ \ 4\ \ 3\ \ 2\ \ 1\ \ 0}_{\text{7-bit ASCII code}}}\quad \text{a byte}$$

Bit 7 is not used, In most cases it will be clear. Appendix D shows all the characters and their ASCII codes in hexadecimal.

To find the ASCII code of a character, M for example, locate M in Appendix D. The least significant four bits is the hexadecimal digit at the left of that row and the most significant three bits is the hexadecimal digit at the top of that column. This gives the ASCII code for M as $4D. The two leftmost columns of characters in Appendix D are **control codes**. The control codes are not printable but are used to control an external device that data is being sent to. For example, the control codes FF (form feed) and CR (carriage return) would be used when sending data to a printer. (All the control codes are listed in more detail in Appendix D.)

Character representations are stored in memory as a series of bytes, each byte containing the ASCII code of a character. An illustration of characters stored in memory at sequential addresses $2000 through $2014 follows. This example can be decoded by using Appendix D.

```
2000   54 68 69 73 20 69 73 20
2008   41 53 43 49 49 20 65 6E
2010   63 6F 64 65 64
```

## DATA STRUCTURES

When storing large amounts of data in memory, it must be organized in some way. Organization methods have been developed to help keep track of where the data is and to facilitate its use. Each method has limitations and advantages. The use of data organization methods in memory is the basis of **data structures.** Normally data are stored in groups where all the data in one group are of the same type. For example, one group may be composed of wind speeds at a certain geographic location and another group may be composed of wind speeds at another geographic location. The individual wind speeds are the **elements** of that group. If the elements are not in any particular order in the group, then that group is a **block** of data.

### Tables

A **table**, or **list**, is a group of the same type and same length elements where the elements are stored in some order in sequential memory locations. An example would be a table of wind speeds for a year at your house. Each element would represent the wind speed for one day and be composed of one byte. The first element of the table is the wind speed for January first, the second is the wind speed for January second, etc. Here the order is chronological, but it could be alphabetical or numerical. Since each element of the example is composed of one byte, the first element would be at the first memory address of the table and the second element would be at the second memory address, etc. The table can be represented as:

```
5000   E0 E1 E2 E3
5004   E4 E5 E6 E7
            .
            .
```

The table starts at address $5000 and each element is denoted by EX, (X indicates the element number).

The starting address of, or **pointer** to, the table must be known. The programmer knows the pointer value when the table is first being built, but the pointer must be saved so the table can be accessed later. It is also a good idea to note the length of the table, or its ending address, and how many bytes each element uses. This information can be stored at the very beginning of a table, before the data elements. One way would be to store the number of elements in the first byte of the table and to store the number of bytes each element uses in the second byte. Immediately following this would be the elements. Upon initially accessing a table, the number of elements and the length of each element would be read. The user could then calculate the address of any element, and the last address of the table.
The address of any element(X), where the elements are numbered starting with zero, is found with the following equation:

X = element number
L = number of bytes in a element
element address = pointer + 2 + ( X x L )

where 2 is the number of bytes at the beginning of the table that contain the number of elements and number of bytes in an element. The last address used by the table is found through the following equation:

N = number of elements in table
L = number of bytes in a element
last address = pointer + 1 + ( N + 1 ) x L

A table of seven elements with each element three bytes long is illustrated in Fig. 2-4. The pointer is the value $4300, the starting address of the table. Knowing the pointer value, the number of elements, and the number of bytes in each element, the address of any element can be found. The address of the first element, element 0, can be found as follows:

pointer = $4300 N = 7
L = 3 X = element number

address of element #0 = pointer + 2 + ( X x L )
= 4300 + 2 + ( 0 x 3 )
= 4302

You should be able to use this equation to find the address of any element in the table and then double check it against Fig. 2-4. You should also be able to determine the last address used by this table. Remember when doing the arithmetic that the numbers are hexadecimal.

```
                         _ele #0_  _ele #1_
4300      NN   LL   ::  ::  ::  ::  ::  ::

              _ele #2_  _ele #3_  _ele #4_
4308      ::  ::  ::  ::  ::  ::  ::  ::

          __  _ele #5_  _ele #6_
4310      ::  ::  ::  ::  ::  ::  ::
```

Fig. 2-4 A Table in Memory.

**Linking Tables**

Next we need to know how to use two or more of the same type of tables residing in memory concurrently. A pointer for each table would allow access to any of the tables in any order. Often tables would be read or used in a specific order; take the case of three annual tables of daily rainfall in Boston. There could be a table of the rainfall for the year 1980, the year 1981, and the year 1982. After scanning a table in chronological order, one would have to go back and find the pointer for the next year's table. This process could be simplified if the tables were somehow **linked**. The tables can be linked by placing the pointer to the next table at the end of the current table, this is illustrated in Fig. 2-5.

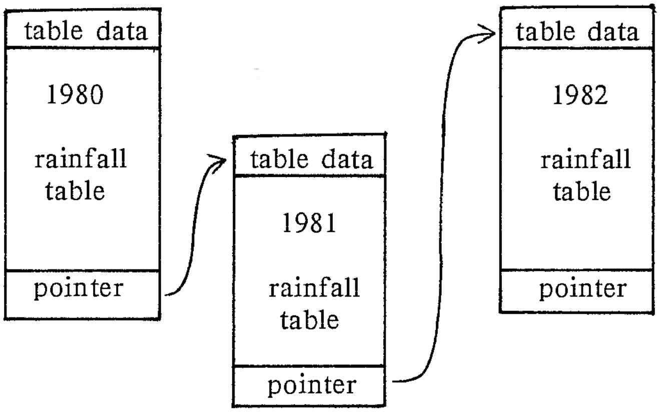


Fig. 2-5 Linked Tables.

In Fig. 2-5 each table is made up of three components: the table data which gives the number of bytes in each element and the number of elements in that table; the body of elements; and a pointer which is the address of the next table. The pointer of the very last table would have to contain a unique code, such as $FFFF, to indicate there are no more tables. That pointer could also point back to the first table, resulting in a set of circularly linked tables. The linkage in Fig. 2-5 can be made circular by drawing an arrow from the pointer of the last table to the top of the first table. Circularly linked tables would be used when repetitively searching the tables for certain elements.

Tables can be **doubly linked** to allow reading either forward or backward. This would require another pointer in each table, to the address of the previous table, as seen in Fig. 2-6.

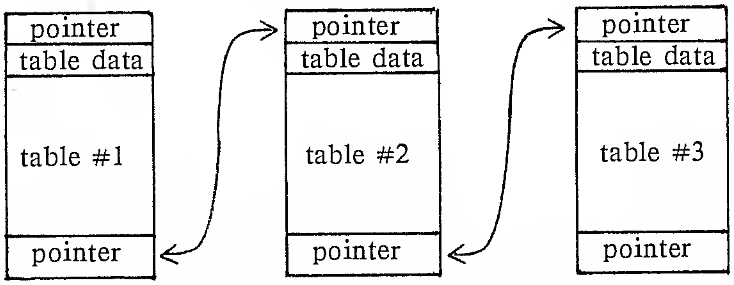


Fig. 2-6 Doubly Linked Tables.

The backward pointer of the first table should contain a unique code, such as a $0000, to indicate there are no tables previous to it. Doubly linked tables can also be circularly linked.

**Directories**

If we were working with a number of related tables, such as rainfall, snowfall, and wind speed, an aditional technique is needed to keep track of them. A **directory** is used; the directory is a table of pointers to the related tables. In Fig. 2-7, the directory is composed of three elements: the three pointers to the to the Boston weather tables.

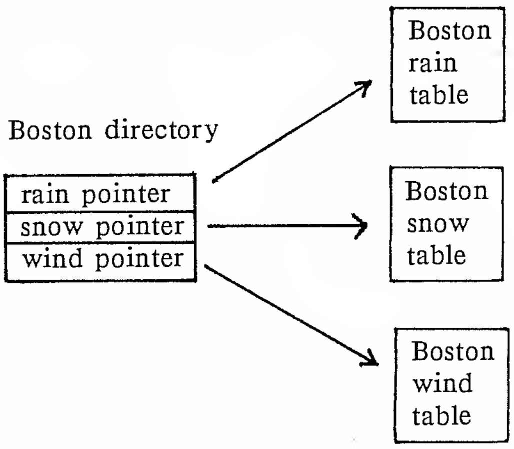


Fig. 2-7 A Single Level Directory.

Were weather tables available for many cities, Fig. 2-7 would be expanded to include a directory of cities to point to each city's directory. Two levels of directories can be seen in Fig. 2-8. This system of directories pointing to sub-directories is called a **tree directory**.

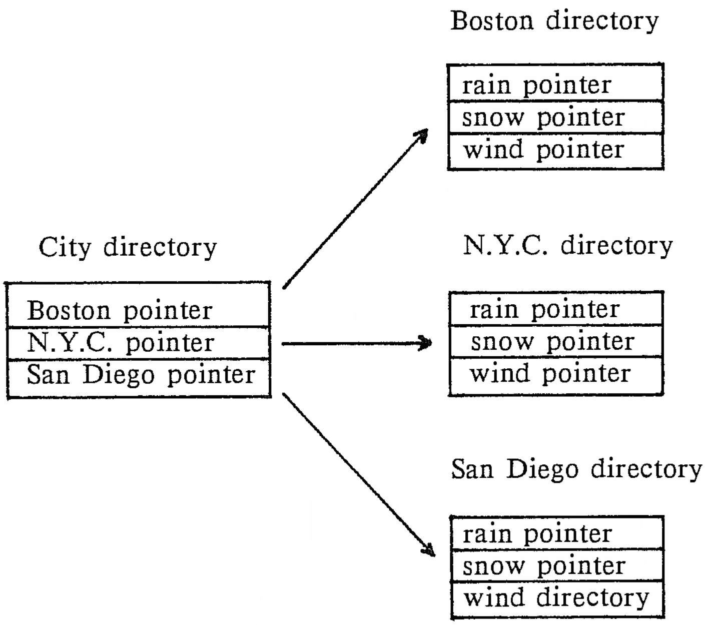


Fig. 2-8 A Two Level Directory.

Directories are also used to keep track of where information is stored on a disk. In fact, the most important table stored on a disk is the disk directory.

**Queues**

Another method of organization is a **queue**. A queue is composed of a list of elements in sequential memory locations and a pointer to the next element to be read. One type of queue is the **first-in-first-out** or **FIFO**. A FIFO queue is analogous to a waiting line: the first person in the line is the first to be served. In a FIFO queue the first, or oldest, element is found by a pointer that contains that element's address. After the first element is processed the pointer is directed to the next oldest element in the queue.

Another type of queue is the **last-in-first-out**, or **LIFO**. This is analogous to a stack of blocks where the first block into the stack is at the bottom and the most recent block put on the stack is at the top of the stack. In fact, a LIFO queue is also known as a **stack**. When removing blocks from the stack, the first lifted off is the top and most recently added block. A LIFO queue is organized as elements stored in sequential memory locations. A pointer contains the address of the most recently added element. After that element is read, or taken off the stack, the pointer is changed to point to the next most recently added element. To add an element to a stack is to **push** it onto the stack, and to retrieve an element from a stack is to **pull** or **pop** it off the stack.

## EXTENDED COLOR BASIC DATA FORMATS

Extended Color BASIC can work with numeric and string, or text, variables. The computer has an internal format for these variables and an organizational scheme for storing them in memory. To write assembly language programs using BASIC variables, we need to know their location and format.

To find were BASIC has stored a variable that variable's pointer must be found. The pointer to variable X is found with the VARPTR(X) BASIC function. However, before the pointer can be found the variable must have been previously used or established by BASIC. The following commands will produce the pointer of variable X:

```
X = 1234
PRINT VARPTR(X)
```

The displayed value will be a decimal address. Remember that the numeric values BASIC accepts or prints are normally decimal. The value of a pointer can also be assigned to another variable. This can be done with the following commands:

```
X = 1234
A = VARPTR(X)
PRINT A
```

### Numeric Variables

A variable's value is represented in five bytes in binary floating point format. The pointer of a numeric variable, obtained by using the VARPTR function, will be the decimal address of the first byte. The binary floating point numbers used by Extended Color BASIC have a normalized mantissa composed of 32 bits, or four bytes. The floating point number can be written as:

$$+/- .BBBBBBBB * 2^{+/-E}$$

where each B represents a nibble. The two quantities that completely describe the floating point number are the mantissa and the exponent. The quantities are stored in five bytes in sequential memory locations:

EE BB BB BB BB

The first byte contains the exponent and the last four bytes contain the mantissa.

The exponent, which can be either positive or negative, is encoded in the first byte with a modified binary representation. The decimal value of the exponent is found by subtracting a decimal 128 from the decimal value of the exponent byte. Thus, if the decimal value of the exponent byte is greater than 128, the exponent is positive, and if the value is less than 128, the exponent is negative. Of course, if the exponent byte's value is 128, the exponent is zero. Some examples are:

Exponent byte=$A2_{16}=162_{10}$,
so exponent = $162-128 = +34_{10}$.

Exponent byte =$56_{16}=86_{10}$,
so exponent = $86-128 = -42_{10}$.

The hexadecimal value of the exponent can be found by subtracting $80 from the hexadecimal value of the exponent byte. An example of the hexadecimal calculations can be seen below:

Exponent byte = $88,
so exponent = $88 - $80 = +$8

The exponent is positive if the MSB of the exponent byte is set, and negative if it is not set. The exponent byte of a value of zero is reserved for a special purpose. If the exponent byte is zero, the number as a whole is considered to be zero.

The four bytes representing the mantissa are in another modified binary format. The sign of the mantissa is indicated by the MSB of the most significant byte. If that bit is set, the mantissa is negative, and if it is clear, the mantissa is positive. Considering that the mantissa is normalized, the first bit to the right of the binary point is always set. Since this is so, that bit can always be assumed to be set and it does not have to be explicitly represented in the four byte code of the mantissa. The remaining bits, to the right of the sign bit correspond to the bits to the right of the always set bit of a normalized mantissa. An example of decoding the hexadecimal four-byte mantissa field into the actual binary mantissa is:

```
4 bytes =  03   41   01   00
binary code = 0000 0011  0100 0001  0000 0001  0000 0000
               \
mantissa  =    + .1000 0011 0100 0001 0000 0001 0000 0000
```

where the underlined .1 is always assumed to be there and the bits from the binary code, after the sign bit, are copied behind it. Now a whole floating point number can be decoded from the format used by Extended Color BASIC. An example follows:

```
5-byte code is    83 C4 20 00 00
the exponent = 83 - 80 = +3
the mantissa is found as:
        C4          20          00          00
   1100 0100   0010 0000   0000 0000   0000 0000
   \
   -.1100 0100 0010 0000 0000 0000 0000 0000
the whole floating point number is;
   -.1100 0100 0010 0000 0000 0000 0000 0000 x 2^+3
```

The binary value can be found by multiplying the mantissa by $2^{+3}$ or by using the technique of shifting the binary point, resulting in: -110.00100001, where the trailing zeros have been dropped. You should be able to convert this to decimal.

**String Variables**

A string variable exists in memory as a sequence of codes representing a sequence of characters. Each code occupies one byte. The code used is ASCII, except for a few special characters. The following BASIC program will print many of the text characters and their codes.

```
10 CLS:X=32
20 FOR K=0 TO 5
30 PRINT HEX$(X+16*K);" ";CHR$(X+16*K);" ";
40 NEXT K
50 X=X+1
60 IF X<48 THEN PRINT:GOTO 20
70 GOTO 70
```

This program will put six pairs of columns on the screen. The left column of a pair is the hexadecimal code and the right column is the character. (A complete listing of the text characters and their codes can be found in Appendix C.)

The pointer, given by the VARPTR function of a string variable, is the decimal address of the first byte of the **string descriptor.** (Remember that the string variable must have been previously used by the BASIC program.) The string descriptor is a group of five consecutive bytes in memory that describe a particular string. The first byte contains the number of characters in the string in straight binary. The third and fourth bytes contain the address of the first byte of the string. The second and fifth bytes are used by BASIC and should not be changed. A diagram of the string descriptor is:

```
NN XX AA AA XX
```

The NN byte contains the length of the string, and AAAA is the starting address of the string. You can use the PEEK function to inspect the string descriptor or the string itself at address AAAA.

**Numeric Variable Arrays**

Extended Color BASIC can use single dimension arrays, A(X), two dimension arrays, A(X,Y), and three dimension arrays, A(X,Y,Z). Each element of an array is a numeric variable stored in memory just like a regular numeric variable, that is, as a five-byte binary floating point number.

The pointer to the first byte of an array element is obtained by using the VARPTR function. However, the pointer to the first element of an array is the value given by VARPTR plus seven. For a single dimension array that has already been dimensioned via the DIM command, the pointer can be found as seen below:

```
10 DIM A(9)
20 P = VARPTR(A(0)) + 7
30 P2 = VARPTR(A(2))
```

P equals the starting address of element 0 or the whole array. P2 is the starting address of element 2.

Each numeric variable of a single dimension array will occupy five consecutive memory locations. The next variable follows immediately after each variable. A single dimension numeric array is arranged in memory starting at address P as seen below:

```
          P      A(0)
          P+5    A(1)
     P2 = P+A    A(2)
          .      .
          .      .
```

The pointers to the elements of a two dimension array are found as follows:

```
10 DIM A(2,2)
20 P = VARPTR(A(0,0)) + 7
30 P2 = VARPTR(A(2,0))
```

In the two dimension array, A(2,2) for example, elements are arranged in memory as follows:

```
          P        A(0,0)
          P+5      A(1,0)
     P2 = P+A      A(2,0)
          P+F      A(0,1)
          P+14     A(1,1)
          P+19     A(2,1)
          P+1E     A(0,2)
          P+23     A(1,2)
          P+28     A(2,2)
```

The pointers to elements of a three dimension array are found as follows.

```
10 DIM A(1,1,1)
20 P = VARPTR(A(0,0,0)) +7
30 P4 = VARPTR(A(0,0,1))
```

The elements of array A(1,1,1) are arranged in memory starting at address P as follows:

```
          P        A(0,0,0)
          P+5      A(1,0,0)
          P+A      A(0,1,0)
          P+F      A(1,1,0)
     P4 = P+14     A(0,0,1)
          P+19     A(1,0,1)
          P+1E     A(0,1,1)
          P+23     A(1,1,1)
```

**String Variable Arrays**

String variable arrays of single, two, and three dimensions are also available. The string elements of an array are arranged in an orderly fashion in memory, but because strings may be of different lengths, it is not easy to calculate the address of any other string by knowing the address of just the first string element. It is best to find the pointer to the string of concern.

The pointer to a string in an array is found like that of a string variable. First all the string elements of the string array must have been used by the BASIC program. Using them establishes their position in memory. The value given by the VARPTR function is the address of the string descriptor, where there is a string descriptor for each string in the array. However, the pointer to the string descriptor of array element 0 is the value given by VARPTR plus seven. Again, the first byte of the string descriptor is the length of that string, and the third and fourth bytes contain the address of the first byte of that string. The pointer to the string descriptor can be found as follows:

```
10 DIM A$(9)
15 FOR X=0 TO 9
20 A$(X)="STRING" + STR$(X)
25 NEXT X
30 P=VARPTR(A(3))
```

P is the address of the string descriptor of array element A$(3). The address of the first byte of this string can be calculated and printed, in hexadecimal, by adding the following

```
40 P1=256*PEEK(P+2)+PEEK(P+3)
50 PRINT HEX$(P1)
```

Statements 15 - 25 use, and thus establish, each string element of the array.

# CHAPTER 3

# Introduction To The MC6809E Microprocessor

The heart of the Color Computer is the MC6809E microprocessor, manufactured by Motorola, Inc. This microprocessing unit, or MPU, exists as a **large scale integrated (LSI)** circuit housed in a forty pin dual-in-line package. The MC6809E is a very advanced eight-bit MPU with features that include ease of programming, powerful instructions and addressing modes, and the ability to perform many eight-bit and some 16-bit operations. The MPU, the major subunit of the Color Computer, can be programmed, or directed, to manipulate data, perform calculations, and control the operation of the other subunits within the Color Computer.

The MPU is directed to perform a series of operations with a program consisting of **machine instructions** and data. The machine instructions are conceptually like BASIC commands, each performs a unique operation on some variable. The machine instructions to be executed by the MPU must be in memory, usually in sequential memory locations. An instruction consists of one or more bytes, and the first byte is the **operation code**, or **op code**. The operation code is a binary coded command that directs the MPU to perform a certain operation. Immediately following the op code is the **operand field**. The operand field may contain the data or the address of the data to be operated upon. The data to be operated on is the operand. The general format of an instruction with a one-byte op code and a two-byte operand field can be shown in memory starting at address $2000 as:

```
2000   XX YY YY
```

XX is the op code and YYYY occupy the operand field.

An example of an MC6809E instruction is the Increment instruction, with an op code of $7C. This op code will direct the MPU to increment (add one to) the binary value stored at the two-byte address immediately following the op code. This instruction would appear in memory as seen below.

```
$2000   7C 05 CD
```

After executing the instruction, the byte stored at address $05CD will have been incremented by one.

Building a program by assembling op codes, addresses, and data is known as **machine language** programming or **machine coding**, since the programmer works intimately with machine instructions. The programmer continually looks up the op codes and lengths of various instructions, addresses of where data are stored, and then, by hand assembles the op codes and operand fields to form the instructions of a program. This is termed the lowest level of programming. It is quite tedious, and in the process of working with the program particulars it's easy to become divorced from the big picture of the problem the program is meant to solve.

Assembly language is the next higher level language, though it is still low level. The programmer is assisted by a program known as an **assembler**. The programmer specifies each instruction by its **mnemonic**, or shorthand English name. For instance, the mnemonic of the Increment instruction is INC. One can also give a particular group or field of bytes a **label**, or **name**, that will represent its starting address. Thus an assembly language statement could appear as:

```
INC  COUNT
```

where COUNT is the label of the byte to be incremented. The assembler would, on receiving this statement, assemble the op code and operand to construct an increment machine instruction.

Whether one is programming in machine or assembly language, the internal structure and operation of the MPU must be understood, to be aware of the operations that can be performed, what limitations exist, and how to direct the operations to most efficiently (time wise and/or memory usage wise) arrive at a desired result.

## MC6809E INTERNAL ARCHITECTURE

Before investigating the internal structure of the MC6809E we will look in more detail at the interconnections between the MPU and the memory. In Fig. 3-1 the MPU is connected to the memory by three data paths, collectively known as the **system bus**. Each path or bus is no more than a group of wires that conduct or transmit the state of a bit, one bit per wire, along its length.

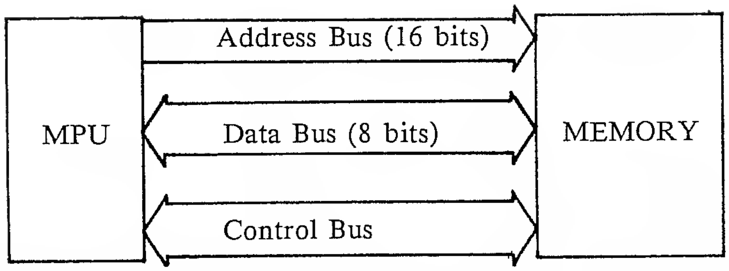


Fig. 3-1 The Interconnection of MPU and Memory.

The **address bus** is composed of 16 wires to transmit a 16-bit address to the memory. This technique of data transmission, where all the bits are sent simultaneously, each on its own wire, is known as **parallel** data transmission. As you can see in Fig. 3-1, the address bus is unidirectional; the address is only sent from the MPU to the memory. The **data bus** is composed of eight wires for transmitting one byte at a time. This bus is bidirectional because the MPU could send a byte to be stored in memory or receive a byte read from memory.

The **control bus** is composed of a number of special purpose bits or control signals. One bit is the $R/\overline{W}$ signal to tell the memory to read ($R/\overline{W} = 1$) a byte from the address on the address bus and send it to the MPU on the data bus. This signal could also indicate a write ($R/\overline{W} = 0$) operation of the byte on the data bus into the location on the address bus. Other bits may indicate whether the MPU is using the system bus or if the MPU is running. The control bits will be covered in detail later in this chapter.

Fig. 3-2 shows the major components within the MC6809E and how they are connected. The most numerous component in the MPU is the **register**. A register is a group of electronic circuits in which eight or 16 bits can be stored. The registers are similar to memory cells in that they can store binary data. Unlike memory cells, the registers are not specified by an address, but by their name or function. Each register serves one or a limited number of purposes. Throughout the following discussions, various MC6809E instructions will be casually introduced to clarify the use of the internal components. A detailed description of each instruction can be found in Chapter Five.

## Controller

The **controller** block is the device that orchestrates the internal operations to accomplish some particular task. Tasks can include reading an instruction from memory, decoding an instruction's op code, and directing the other components to perform an instruction. The instruction (or I) register within the controller is where the op code of the current instruction is stored. The controller decodes the op code to determine the actions to be taken to perform one instruction. Other results of decoding the op code are the length of the instruction and whether the bytes immediately following the op code are data or the address of the data to be manipulated.

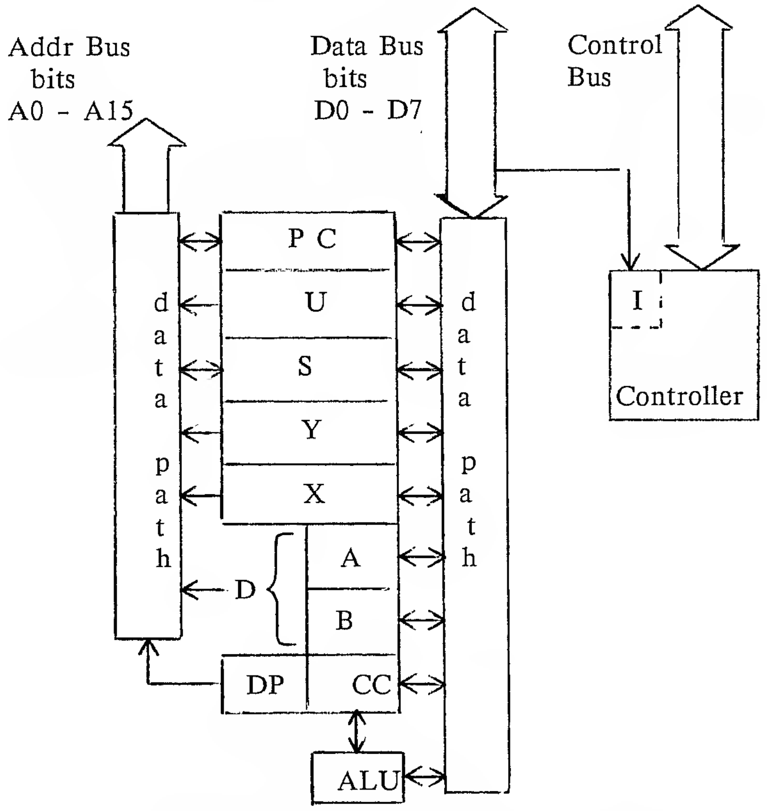


Fig. 3-2 A Block Diagram of the MC6809E MPU.
Courtesy Motorola,Inc.

**Program Counter**

The **program counter** (or **PC**) register is a 16-bit register containing the address of the next instruction to be executed. For instance, to start executing a program that starts at address $3000, the program counter register would be loaded with $3000, the address of the first byte of the first instruction of that program. At this point the controller would initiate the **instruction fetch** sequence by sending the content of the program counter register out on the address bus and telling the memory it wants to read a byte from that address. Shortly thereafter the first byte of the instruction, the op code, would be transferred from memory to the MPU on the data bus and directed into the instruction register in the controller. Then the controller would direct the program counter register to be incremented by one. This process is repeated until the entire instruction, the op code and operand field, has been read out of memory and into the MPU. After reading one complete instruction, the PC register would have been incremented to point to the first byte of the next instruction. This is why it is important that instructions reside sequentially in memory.

## Arithmetic Logic Unit

The **arithmetic logic unit**, or **ALU**, is a collection of electronic circuits that perform the arithmetic and logic operations. The ALU can be conceptually represented as shown in Fig. 3-3. It can accept up to two bytes, perform an operation on them, and produce a new byte. Usually one or two bytes are sent to the ALU; the controller then directs the ALU to perform a specific operation them, then routes the resulting byte to its destination. The arithmetic operations that can be performed are addition, subtraction, incrementing, decrementing, decimal adjust, multiplication, and sign extension. The logic operations that can be performed are AND, OR, XOR, shift, rotate, and complement.

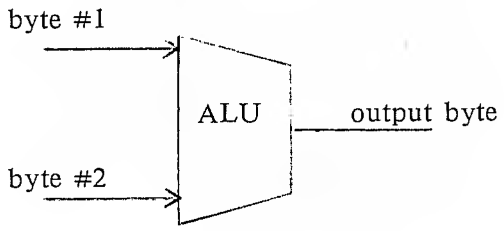


Fig. 3-3 The Arithmetic Logic Unit (ALU).

An example of using the ALU is demonstrated with the Complement instruction. In this case the instruction will exist in memory as:

```
$1000   73 2F 02
```

where the content of $2F02 is $FA. $73 is the op code of the Complement instruction, and the byte to be complemented is at address $2F02. First the instruction will be fetched from memory; then the controller will direct a read operation from memory address $2F02. The byte read is routed to the ALU and the controller directs the ALU to perform the complement operation. The resulting byte, the complement of the original byte, is then routed from the ALU in the MPU on the data bus to the memory. Finally, a write operation is initiated to store the byte on the data bus at address $2F02. The instruction is now complete and address $2F02 contains $05, the complement of $FA. Also, the PC register has been incremented to point to the next instruction and the instruction fetch operation is beginning again.

## Condition Code Register

The ALU performs another very important function; setting or clearing certain bits, or **flags**, in the **condition code**, or **CC**, register. The bits of the condition code register are modified by the ALU to indicate the general outcome of an arithmetic or logic operation. The condition code register is an eight-bit register; each bit, or flag, is an indicator of some previous event or condition. The condition code register is organized and each bit labeled as shown below.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | bit positions |
| E | F | H | I | N | Z | V | C | bit labels |

The C, V, Z, N, and H bits are set or cleared by the ALU to serve as a record of the general outcome of the operation. The E, F, and I bits are set or cleared under other conditions to later indicate the conditions of the MPU. The E, F, and I bits will be described in detail later in this chapter.

The C bit, bit 0 of the condition code register, indicates whether a binary carry was generated by the ALU beyond its eight working bit positions. If a carry is generated from the ALU during binary addition, the C bit is set; otherwise it is cleared. Two examples of binary addition are below.

```
      c  c c c c  c                          c
         1 0 1 1  0 1 1 0           1 0 1 1  0 0 1 0
       + 0 1 1 0  1 1 0 1         + 0 1 0 0  1 0 1 1
 C=1     0 0 1 0  0 0 1 1   C=0     1 1 1 1  1 1 0 1
```

In the lefthand example a carry was generated from the most significant bit position, so the C bit is set. In the second example no carry from the MSB position was generated, so the C bit is cleared.

After a subtract operation, the C bit will be set if a borrow was generated from the ALU. This process can be seen as:

```
                b                   b            b
         1 0 1 1  0 1 1 0           0 0 1 1  0 1 1 0
       - 0 0 1 0  1 0 1 0         - 1 0 0 1  0 1 0 1
 C=0     1 0 0 0  1 1 0 0   C=1     1 0 1 0  0 0 0 1
```

In the lefthand example, a borrow was not generated from the ALU, so the C bit is cleared. In the righthand example, a borrow was generated from the ALU, so the C bit is set.

The V bit, bit 1, is set if an overflow or underflow condition is generated when adding or subtracting signed binary numbers. In particular, the V bit is cleared if the two leftmost bit positions either do or do not both generated a carry. The V bit is set if only one of the two leftmost bit positions generate a carry. Two examples of this are:

```
             * *                     * *
           c c      c                c c c    c c
             1 1 0 0  0 1 1 0        0 1 0 1  0 0 1 1
           + 1 1 1 1  0 1 0 0      + 0 0 1 1  0 1 1 0
      V=0    1 0 1 1  1 0 1 0  V=1   1 0 0 0  1 0 0 1
```

where the asterisks mark the positions of the two leftmost bits. In the lefthand example, a carry was generated by bit 6 and bit 7, so there is no overflow and the V bit is cleared. In the righthand example, a carry is not generated by bit 7 but a carry is generated by bit 6. Therefore there is an overflow, and the V bit is set to indicate this. You should be able to verify that the addition result is incorrect in the righthand example.

The Z bit, bit 2, will be set if the result of the operation is zero; that is, if all eight bits of the resulting byte are zeros.

The N bit, bit 3, will be a copy of the MSB of the resulting byte. This is a useful indicator when working with signed binary numbers, since the MSB of the number, and hence the N bit, indicate the sign of the number. If the N bit is set, the resulting signed binary number is negative.

The H bit, bit 5 of the condition code register, indicates whether there was a carry from bit 3 of the ALU after an eight bit addition. The H bit will be set if there was a carry and it will be clear of there was no carry. The H bit essentially indicates if there was a carry from the least significant nibble to the most significant nibble. Two examples of this are:

```
                      *   c              c     c *
             0 1 1 1  1 0 0 1            0 1 0 0  1 0 1 0
           + 1 0 0 0  0 1 0 1          + 0 1 1 0  1 1 0 0
      H=0    1 1 1 1  1 1 1 1    H=1     1 0 1 1  0 1 1 0
```

where the asterisk marks the position of bit 3, or the source of the **half-carry**. In the lefthand example no carry was generated by bit position three so the H bit equals zero. In the righthand example there is a carry from bit position three so the H bit equals one. After a subtraction, the value of the H bit is not defined; that is, it is not set to any particular state. The H bit is used by the MPU when performing a decimal adjust on binary coded decimal numbers.

Not all of the MC6809E instructions have an effect on all the condition code bits described so far. The conditions that the bits of the condition code register represent are just not possible or applicable to some instructions. A detailed description of the instructions in Chapter Five will indicate which condition code bits are modified by each instruction.

The condition code register is probably the most important register in the MPU because it contains the information in the C, V, Z, N, and H bits that control the decision-making process. Decisions are made with the **conditional branch** instructions. A conditional branch instruction will branch, or jump, to a specified address (the operand) if the states of particular condition code bits match the states the conditional branch is looking for. If the states do not match, the conditional branch will do nothing and execution of instructions will continue with the next instruction after the conditional branch. A conditional branch that looks for a match, or tests the N bit of the condition code register is the Branch on Minus (BMI mnemonic) instruction. This instruction would appear in symbolic notation as:

```
BMI  $172E
```

If the N bit is set, the MPU will be directed to start executing instructions starting at address $172E. If the N bit is not set the MPU will fall through the branch and continue by executing the following instructions. This instruction can make the decision to continue processing the following instructions or to branch to a different series of instructions at address $172E: if the result of the previous operation set the N bit (the resulting byte has a negative value), then go to $172E. This is similar to the BASIC IF... THEN command. There are 16 different conditional branch instructions that test the various combinations of the C, V, Z, N, and H bits being on of off.

**A, B, and D Registers**

The A and B registers are two identical eight-bit registers that can each hold one byte. Each of the registers is also known as an **accumulator** because they are used in close association with ALU. The results of a series of arithmetic or logic operations accumulate in the A or B register. It is in these registers that a byte about to be manipulated or the resulting byte from the ALU are stored.

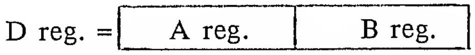
Let's explore some of the capabilities of the A and B registers by using some instructions. Suppose we want to exclusive or the two bytes that reside at addresses $3100 and $3104, and then store the result at address $3108. First we would use the Load A (LDA) instruction to read a byte from memory address $3100 into the A register. This instruction, using its mnemonic, would appear as: LDA  $3100. Then the Exclusive OR A (EORA) instruction could be used. This instruction will read a byte from memory, perform the exclusive or operation between that byte and the byte in the A register, and then route the resulting byte into the A register, destroying the register's previous contents. This instruction will appear as: EORA

$3104 .We now have the answer we want, but it is in the A register. It can be stored in memory with the Store A (STA) instruction as: STA  $3108. The Store A instruction will store the contents of the A register at memory address $3108. This program would collectively reside in memory as shown below.

```
LDA  $3100
EORA  $3104
STA  $3108
```

The same, or a similar sequence of operations, can also be performed using the B register:

The A and B registers can also be referred to as a single entity, the D register. The D register is 16 bits long; the upper eight bits are the A register and the lower eight bits are the B register. This relationship is illustrated below.



A use of the D register is the Store D (STD) instruction. This instruction stores the most significant byte of the D register in memory at address N and the least significant byte at address N+1. For example: STD $2000 will store the contents of the A register at $2000 and the contents of the B register at $2001.

**X and Y Registers**

The X and Y registers are two identical 16-bit registers. Each register is primarily used to hold an address, but can also be used to hold a double byte of data. These registers are known as **index registers** because they are typically loaded with the starting address of a table and used to generate the pointer, or index, to elements within that table. A limited number of arithmetic operations can be performed on the contents of the index registers. One operation is the addition of a signed constant number, whose value can be from -32768 - +32767 decimal, to the contents of an index register. Another is the addition of the contents of the A, B, or D registers, in signed binary format, to the contents of an index register. Finally, the **auto-increment** and the **auto-decrement** functions will increment or decrement the contents of an index register by a value of one or two. The results of these operations are all routed back into the original index register.

A similar group of arithmetic operations can be performed on the contents of an index register to generate an **effective address**. The effective address is the actual address sent on the address bus to the memory to specify a particular byte to be accessed. These operations will not alter the contents of the index register; they only generate the effective address. One operation is to add to the value in an index register a signed constant number whose value can be from -32768 - +32767 decimal. The other operation is to add to the value in an index register the contents, in signed binary format, of the A, B, or D register. These operations let us preserve the starting address loaded into the index register.

When using an index register as a pointer, that index register will be in the operand field of the instruction. A Load A instruction that uses the Y register as the pointer to the desired data would be written in symbolic format as: LDA ,Y. This will load the A register with the byte stored at the address contained in the Y register. If the Y register contains the starting address of a table, the effective address of a particular byte can be calculated within the operand. Examples are:

```
LDA  10,Y
LDA  B,Y
```

The first example will load the A register with the byte from the effective address, calculated as the sum of the contents of the Y register and 10, decimal. The second example will load the A register with the byte from the effective address, calculated by adding the contents of the Y register to the contents of the B register. In neither example will the Y or B registers be changed. If the Y register points to the end of a table, the following instruction can be used to read a byte from that table: LDA -42,Y. In this case the effective address will be the contents of the Y register minus a decimal 42.

The operations that modify the contents of an index register or generate an effective address let one rapidly and easily calculate the address of an element in a table. Also, since the index registers are 16 bits long, any location within a 64K memory may be accessed. This is one feature that makes the MC6809E MPU a powerful microprocessor.

## U and S Registers

The U and S registers are two identical 16-bit registers used as index registers or **stack pointers**. When used as index registers, the U and S registers can perform all the functions of the X and Y registers. This means one can use up to four separate index registers. As stack pointers, the U and S registers contain the address of the item at the top of a stack.

A stack, as implemented by the MC6809E MPU, exists as a series of sequential memory locations and the bytes contained within them. The byte at the top (the newest byte) of the stack is at the lowest memory address of the stack. The stack pointer in use, U or S, will contain that address. The byte at the bottom, the oldest byte, is at the highest memory address of the stack. Fig. 3-4 shows the U stack as five bytes in memory; the top of the stack is at address $3FFC and the U register contains $3FFC. The S stack uses the S register as its pointer.

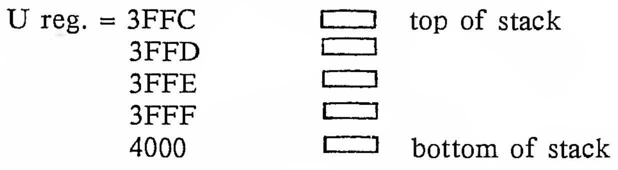


Fig. 3-4 A Stack in the MC6809E

When a byte is pulled from a stack, the top byte is read and then the stack pointer is incremented. The stack pointer is incremented to point to the next byte in the stack and to indicate that the previously read byte is no longer on the stack. This can be done with a Load A instruction, with the U register as the operand and auto-incrementing by one specified. This instruction would appear as: LDA ,U+. This instruction will load the A register with the byte at the address in the U register and then increment the contents of the U register by one. Now the U register will point to the next lower byte in the stack.

A byte can also be pushed, or added, to the stack with a store instruction. The operand will be the U register, with auto-decrementing of one specified. Let's say the U register contains $3FFC, that it points to the top of the stack in Fig. 3-4. The instruction: STA ,-U will first decrement the contents of the U register by one. (Note that the minus sign comes before the U in the operand.) The U register will now contain $3FFB. Then the contents of the A register will be stored at that address, or at the new top of stack.

There are four special instructions for stack use. The PSHU and PULU instructions respectively push on and pull off the U (user's) stack the contents of a register or registers. The PSHS and PULS instructions respectively push on and pull off the S, or hardware, stack the contents of a register or registers. An example of using the PULU instruction is: PULU X , which will load the X register with the two top bytes of the stack and increment the U register by two.

The stack pointed to by the S register is the hardware stack. There are times when the MPU itself will push or pull data to or from this stack in response to certain stimuli. One should not use the S register until more is known about the hardware stack.

**Direct Page Register**

The **direct page**, or **DP**, register is an eight-bit register that holds a byte representing the upper eight bits of the operand address. This capability is used when **direct addressing** is specified. First the page register must be loaded with an appropriate value. Then instructions using direct addressing may be used. Assume the DP register has been loaded with $23. An instruction using direct addressing will appear as STA <$7C. This instruction will store the contents of the A register in address $237C. You can see the content of the DP register is linked, or **concatenated**, with the single byte after the op code to form the effective address. This technique allows for shorter instructions so more instructions will fit in a given amount of memory, and the instructions will execute faster since fewer bytes must be read from memory.

The internal components of the Mc6809E microprocessing unit are the items with which a programmer must work. Some of the components, such as the ALU, controller, and data paths, are not explicitly controlled by the programmer, but they do support the execution of various instructions. As such, their operation is transparent. For this reason the diagram most used as a programming aid is the simplified version of a block diagram shown in Fig. 3-5. The programming model in Fig. 3-5 depicts all the controllable registers and labels the bits of the condition code register.

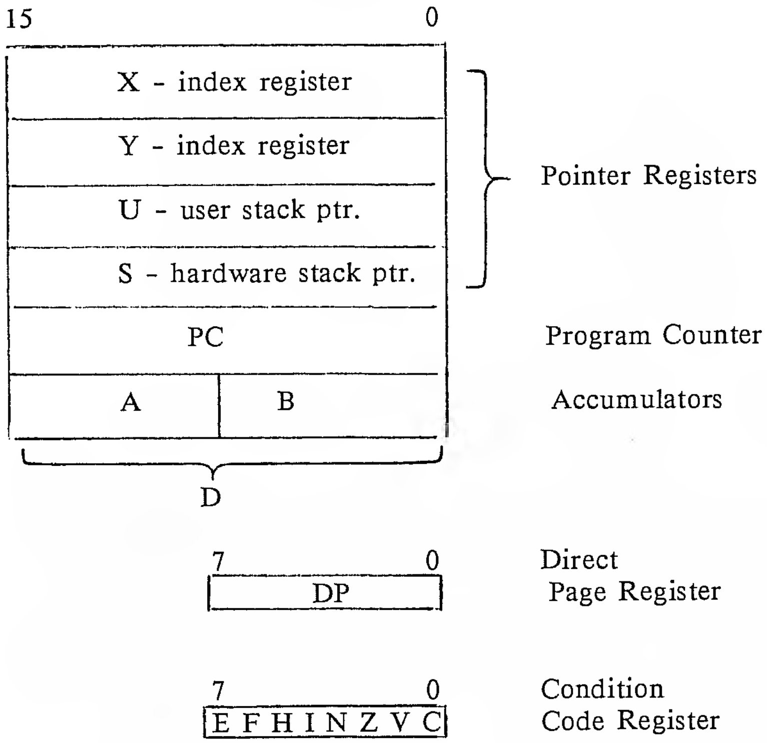


Fig. 3-5 A Programming Model of the MC6809E
Courtesy of Motorola,Inc.

## MC6809E EXTERNAL CONNECTIONS

Data, addresses, and control signals are electrically transmitted to or from the MPU via conductive pins that protrude out of the package the MPU is housed in. The dual-in-line package that houses the MPU integrated circuit and the connecting pins is illustrated in Fig. 3-6. The arrangement shown is the view as seen when looking down on the top of the dual-in-line package.

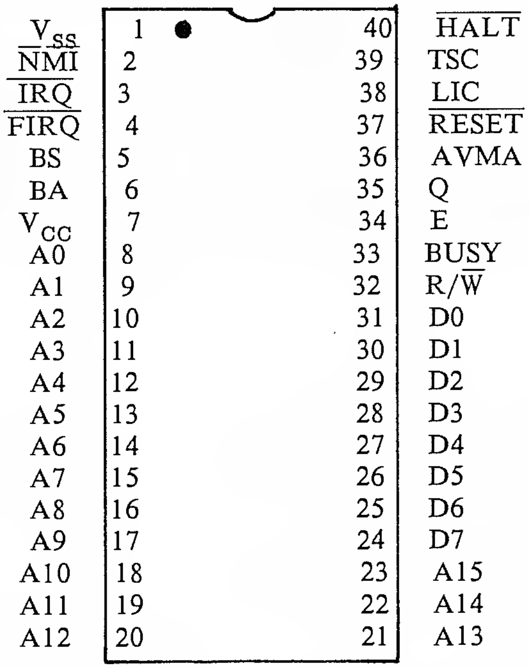


Fig. 3-6 The Pin Assignments of the MC6809E (top view).
Courtesy of Motorola, Inc.

You can see where the address bus (A0-A15) and the data bus (D0-D7) bits enter or leave the MPU as electrical signals. The electrical power is supplied via pins 1 and 7 where pin 1, $V_{SS}$, is connected to ground or a reference and pin 7, $V_{CC}$, is connected to a power source providing +5 volts. The state of a bit is represented by the level of the voltage at its respective pin. A zero bit is electrically defined as being a low voltage of between zero and +0.4 volts. A one bit is a high voltage of between +2 and +5 volts. A zero bit is called a **low** and a one bit is called a **high**. If $82 were on the data bus, pins 24 and 30 would be high and pins 31, 29, 28, 27, 26, and 25 would be low.

The R/$\overline{W}$ (read/write) output, pin 32, is the control signal that initiates a memory read or write operation. A write operation is started by the MPU setting pin 32 low. The byte the MPU put on the data bus is then written in the address the MPU put on the address bus. A read operation is initiated by the MPU setting pin 32 high. A byte is then read from the address the MPU put on the address bus. This byte is sent from the memory to the MPU on the data bus.

The E and Q inputs (pins 34 and 35) to the MPU are the clock signals. The Q clock signal is a series of electrical pulses occurring approximately once every microsecond, or one million times a second. The E clock signal is similar to the Q clock signal except that its pulses occur slightly later. The clock signals are used in the MPU to step, or pace, it through its various sequences of operations.

The $\overline{\text{HALT}}$ input (pin 40) to the MPU will, if a low is applied, stop the MPU upon completion of the currently executing instruction. Putting a high on pin 40 will cause the MPU to resume its operation. The BS and BA output signals, pins 5 and 6, will be set high by the MPU to indicate to other devices on the system bus that the MPU is not running. In the Color Computer the BS and BA signals are not used. While the MPU is halted, it retains the contents of its internal register as long as the E and Q clock signals are present.

The $\overline{\text{RESET}}$ input (pin 37) to the MPU is used to initialize, or prepare, the MPU to run. A reset is performed, pin 37 set low momentarily, when the microcomputer is first turned on or the reset button on the back of the Color Computer is pressed. While the reset signal is held low the DP register is cleared and the F and I bits of the CC register are set. The F and I bits will be described later in this chapter. When the reset signal is set back high, or deactivated, a sequence of operations is initiated. First a byte is read from address $FFFE and loaded into the upper half (bits 8-15) of the PC register. Then a byte is read from address $FFFF and loaded into the lower half (bits 0-7) of the PC register. Then the MPU starts executing the instruction to which the PC register points. The address contained in addresses $FFFF and $FFFE is the reset **vector**, or pointer to the instruction to begin execution after a reset. In the Color Computer, both addresses are in BASIC ROM.

The TSC, BUSY, LIC, BS, BA, and AVMA control signals are not used in the Color Computer. These signals can be used to control sharing the system bus with other devices or MPU's that could be connected to the system bus. A hardware **interrupt** is initiated by $\overline{\text{IRQ}}$, $\overline{\text{NMI}}$, or $\overline{\text{FIRQ}}$ being set active.

## INTERRUPTS

The operation of the MPU can be viewed as a continual fetching and executing of instructions. A program in memory may contain branch instructions causing the MPU to jump from one section of the program to another or cause the MPU to repeatedly execute a certain program section, as in a loop. All operations take place internally with no regard for the outside world. Eventually the MPU must be notified of some external event so it can directed to perform another task. An external event can notify the MPU of its existence by an interrupt. As the name implies, an interrupt can interrupt an MPU that is doggedly executing a program.

An interrupt is generated in one of two ways. A hardware interrupt can be generated by activating one of the three interrupt control signals, i.e. $\overline{\text{IRQ}}$, $\overline{\text{NMI}}$, or $\overline{\text{FIRQ}}$. This is done by momentarily setting one of these signals low. The second type of interrupt is a software interrupt, initiated by one of three machine instructions.

An interrupt directs, or vectors, the MPU to a program that will investigate and/or respond to that interrupt. The initiating process is similar to the reset operation: the MPU will fetch an address, or vector, from a pair of predetermined memory addresses. The vector addresses are stored in BASIC ROM at the 14 highest memory addresses. These addresses are said to be **dedicated** to this purpose. Table 3-1 lists the addresses, their contents, and the function that will use that vector, as set up by Color BASIC 1.1. Table 3-1 is a small part of the memory map of the Color Computer.

| Location of Vector | Vector | Initiator |
|---|---|---|
| FFFE and FFFF | A027 | RESET |
| FFFC and FFFD | 0109 | NMI |
| FFFA and FFFB | 0106 | SWI |
| FFF8 and FFF9 | 010C | IRQ |
| FFF6 and FFF7 | 010F | FIRQ |
| FFF4 and FFF5 | 0103 | SWI2 |
| FFF2 and FFF3 | 0100 | SWI3 |

Table 3-1 The Color Computer Vector Table

If your computer has a different version of BASIC, the vectors may be different than those in Table 3-1. The contents of addresses $FFF2 through $FFFF can be found with the following BASIC program.

```
10 CLS
20 X=0:Y=&HFFFE
30 PRINT"     INTERRUPT VECTOR TABLE"
40 PRINT
50 PRINT"      ADDRESS    CONTENTS"
60 K=256*PEEK(Y-X)+PEEK(Y-X+1)
70 PRINT"      ";HEX$(Y-X);"+";HEX$(Y-X+1);"      ";HEX$(K)
80 X=X+2
90 IF X<14 THEN 60
100 END
```

The program the MPU is vectored to by an interrupt is known as an **interrupt handler** or **interrupt service routine**. After the interrupt handler has completed its task, it may return control, or jump back, to the interrupted program so the program may continue.

## IRQ Interrupt

The IRQ (interrupt request) sequence is initiated if the MPU is running, the I bit of the CC register is not set, and the $\overline{\text{IRQ}}$ pin is momentarily set low. The I bit of the CC register is the IRQ inhibit, or mask, bit. If it is set, the MPU will not recognize an IRQ interrupt.

The sequence will start after the MPU completes the instruction it is currently executing. First the E bit of the CC register is set. This will later indicate that the entire set of registers, except the S register, has been pushed onto the S stack. Then all the registers, the PC, U, Y, X, DP, B, A, and CC, but not S, will be pushed onto the S stack in the order just presented, so the CC register is at the top of the stack. Next the I bit of the CC register is set to prevent any IRQ interrupts until the MPU is again ready to process one. Then a low is sent out on the BA pin and a high is sent out on the BS pin to indicate to other devices on the system bus that the MPU has been interrupted. The upper eight bits (bits 15-8) of the PC register are loaded with the byte read from address $FFF8 and the lower eight bits (bits 7-0) of the PC register are loaded with the byte read from address $FFF9. Then the BS pin is set low. Now the MPU will start to execute the instruction whose starting address was just loaded into the PC register. The Color Computer IRQ sequence of events can be simplified because the BA and BS signals are not used. Its flowchart is shown in Fig. 3-7.

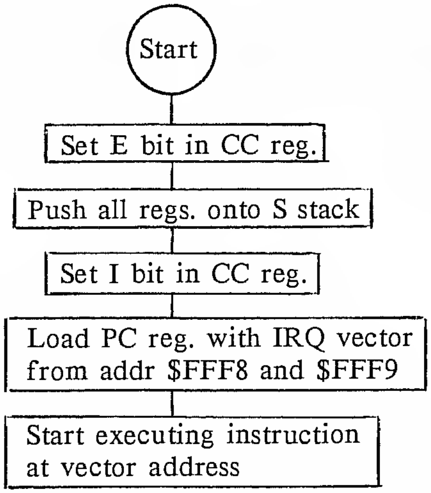

Start
Set E bit in CC reg.
Push all regs. onto S stack
Set I bit in CC reg.
Load PC reg. with IRQ vector from addr $FFF8 and $FFF9
Start executing instruction at vector address

Fig. 3-7 Color Computer IRQ Sequence.

Fig. 3-8 shows the registers in the S stack as the IRQ sequence has stacked them. The subscripted L indicates the lower byte (bits 7-0) of a two-byte register and the subscripted U indicates the upper byte (bits 15-8) of a two-byte register. As you may have guessed, the S register should have previously been loaded with an address at which the stack should start. This is one of the responsibilities of a programmer who will be using interrupts.

after interrupt sequence S -> CC
A
B
DP
$X_U$
$X_L$
$U_U$
$U_L$
$PC_U$
$PC_L$
before interrupt sequence S -> ▭

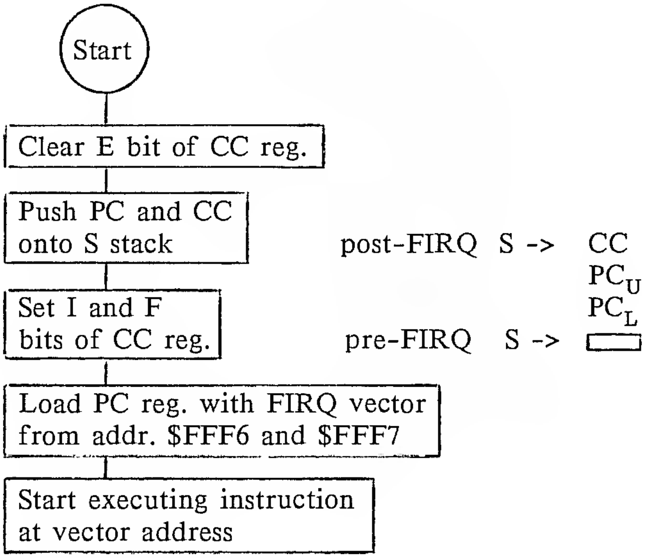
Fig. 3-8 The S Stack after an IRQ Sequence

The interrupt handler program could be a program to interrogate the keyboard to see if any keys have been depressed. During the execution of this program, it is most likely that the contents of the internal registers will be changed. After seeing that no keys were depressed, the interrupt handler program can return control to the point of interruption in the original program. This is done with the Return from Interrupt (RTI) instruction. The RTI instruction will cause the CC register to be pulled off the S stack and its E bit inspected. In the case of returning from an IRQ interrupt, the E bit is set, meaning that all the registers had been pushed into the S stack. If E is set, the RTI instruction will pull the rest of the registers, i.e. A, B, DP, X, Y, U, and PC, from the S stack. The MPU will now resume its operation at the point of interruption with all its registers containing their original values.

**FIRQ Interrupt**

The FIRQ (fast interrupt request) sequence is initiated if the MPU is running, the F bit of the CC register is not set, and the $\overline{\text{FIRQ}}$ pin is momentarily set low. The F bit of the CC register is the FIRQ inhibit, or mask, bit. If it is set, the MPU will not recognize a FIRQ interrupt.

The FIRQ interrupt sequence is very similar to the IRQ sequence. The FIRQ interrupt sequence is faster because only the PC and CC registers are pushed onto the S stack, in that order. During the FIRQ sequence the F and I bits of the CC register are set to prevent another FIRQ or IRQ interrupt before the MPU is ready for one. Also, the E bit of the CC register is cleared to indicate to the RTI instruction that the entire set of registers was not put in the stack. A FIRQ interrupt can interrupt an IRQ interrupt handler program because the IRQ sequence did not set the F bit. In this sense the FIRQ interrupt is of higher priority than the IRQ interrupt. The Color Computer FIRQ interrupt sequence (BA and BS signals not shown) and the resulting stack can be seen in Fig. 3-9. Also note that the FIRQ vector is read from addresses $FFF6 and $FFF7.



Start
Clear E bit of CC reg.
Push PC and CC onto S stack
Set I and F bits of CC reg.
Load PC reg. with FIRQ vector from addr. $FFF6 and $FFF7
Start executing instruction at vector address
post-FIRQ S -> CC
PC_U
PC_L
pre-FIRQ S ->

Fig. 3-9 Color Computer FIRQ Interrupt Sequence

**NMI Interrupt**

The NMI (non maskable interrupt) is the highest priority interrupt; it can not be masked out or inhibited. The MPU is designed so that after a reset the NMI interrupt sequence will not begin until the S register has been loaded. This gives the programmer time to establish where the S stack will start. The NMI sequence is initiated by momentarily providing a low on the NMI pin. The NMI sequence (without BS and BA signals) can be seen in Fig. 3-10. The NMI sequence stacks all the registers, as can be seen in Fig. 3-8.

**Software Interrupts**

A software interrupt can be generated by any of the three software interrupt instructions. Their mnemonics are SWI, SWI2, and SWI3. The software interrupt instructions initiate a sequence very similar to the IRQ sequence. The main difference is that they can not be inhibited. Other differences are: the SWI sequence sets both the F and I bits of the CC register and gets its vector from addresses $FFFA and $FFFB; the SWI2 sequence does not set the I bit of the CC register and gets its vector from addresses $FFF4 and $FFF5; the SWI3 sequence does not set the I bit of the CC register and gets its vector from addresses $FFF2 and $FFF3. You can use Fig. 3-7 to trace their events.

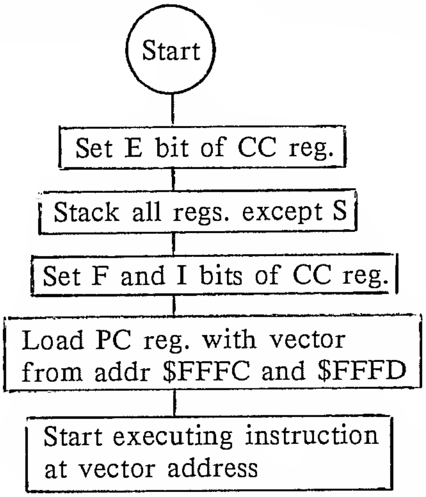

Start
Set E bit of CC reg.
Stack all regs. except S
Set F and I bits of CC reg.
Load PC reg. with vector from addr $FFFC and $FFFD
Start executing instruction at vector address

Fig. 3-10 Color Computer NMI Interrupt Sequence

# CHAPTER 4

# Addressing Modes of the MC6809E

The MC6809E has an instruction set of 59 instructions, of which 40 are manipulative, or perform tests, and 19 are branch instructions. The power of this instruction set is greatly enhanced by the 10 basic **addressing modes** available. Each addressing mode is a different way of specifying where or what is the data to be worked on. The combination of available internal registers, the 59 instruction types, and the 10 addressing modes gives a total of 1464 unique instructions. Fortunately, one does not have to memorize the 1464 combinations to effectively use the MC6809E MPU. Typically, one would specify the required instruction type and then pick an addressing mode suitable for accessing the desired data.

In assembly language a complete instruction is specified by a statement where a statement is made of up to five fields. Each field is a group of character positions in which is put information describing the statement. The fields are statement number, symbol, command, operand, and comment. We'll look first at the command and operand fields, since they are used to specify the instruction type and addressing mode.

Motorola has developed a set of conventions for an assembler to follow in interpreting the various statement fields. The EDTASM+ assembler follows that set of conventions quite closely. The instruction type is specified by putting the mnemonic of the desired instruction in the command field. Sometimes this also specifies the addressing mode; some instructions use only one addressing mode. In general, the addressing mode is specified by the form of the operand in the operand field. These two fields of a statement can appear as:

```
LDA   $2302
```

A space separates the command field (LDA) from the operand field ($2302). The different addressing modes and instruction types will be illustrated by using statements like this one, the way a programmer will specify them. For readers programming in machine code or who want to know each detail, the codes in the machine instructions that determine the addressing modes will also be described.

The 10 addressing modes are register, inherent, immediate, extended, extended indirect, indexed, indexed indirect, relative, PC relative, and direct addressing. Some of the MC6809E instructions can use many addressing modes and some can use only one or a few.

As you know, the total number of unique instructions (1464) is larger than 256, the number of unique op codes that can be represented with one byte. This problem is solved with a **postbyte**. The postbyte is a byte immediately following the machine instruction op code, and it exists only when using the indexed, indexed indirect, extended indirect, PC relative, and register addressing modes. Technically, these addressing modes are all considered to be indexed since they all use a postbyte. The postbyte, when decoded by the MPU, indicates the addressing mode and any particulars of how the operand is to be treated. An instruction, in memory at address 2000, with a postbyte, would appear as:

```
                 op      post     operand
        2000    code     byte      byte
```

## INHERENT ADDRESSING

Inherent, or implied, addressing is the simplest addressing mode; the operand is implied within the op code. A programmer doesn't have to explicitly define the operand by putting something in the operand field of a statement. Some instructions that use inherent addressing are the increment, decrement, clear, negate, and the software interrupts. For example, the Clear A (CLRA) instruction will clear all the bits in the A register. This instruction is composed of just one byte, the op code. The directions to make the MPU clear the A register are contained completely within the op code. Some examples that specify instructions with inherent addressing are: CLRA, INCB, COMA, and SWI. For this addressing mode the operand fields are blank. These examples will cause the assembler to assemble the one byte op codes of the respective instructions. (You can also refer to Appendix B to look up the op codes.) Note that all instructions using inherent addressing are composed of only one byte, except SWI2 and SWI3, which have two-byte op codes. The inherent addressing instructions also execute very quickly, as they use few bytes of memory and work with the internal registers of the MPU.

## IMMEDIATE ADDRESSING

When immediate addressing addressing is specified, the data to be worked with will immediately follow the op code in memory. Immediate addressing is specified in an assembly language statement with the # sign in the operand field. An example would be:

```
LDA  #$4C
```

The Load A (LDA) instruction will load the hexadecimal value of 4C into the A register. This machine instruction would reside in memory at address 1000 as:

```
1000   86 4C
```

$86 is the op code of the LDA instruction with immediate addressing and the following byte is the immediate operand. Some more examples of immediate addressing are: ADDB #$1F and LDX #DEBIT. In the first example, the ADDB instruction will add the value of the immediate operand ($1F) to the contents of the B register, and the result will be routed back into the B register. A symbol is used in the operand field of the second example. The X register is loaded with the address of the symbol, DEBIT. If DEBIT was at address $2102, these machine instructions would appear in memory starting at address 1000 as:

```
1000   CB 1F
1002   8E 21 02
```

$CB is the op code of ADDB with immediate addressing specified, followed by the immediate byte. At 1002 is the op code of LDX with immediate addressing, followed by the two immediate bytes. If the register being used is one byte long, the immediate operand is one byte long, and if the register is two bytes long, the operand is two bytes long. A point to remember is that the data to be used is part of the instruction.

**EXTENDED ADDRESSING**

Extended addressing is specified in a statement by prefixing the operand with a > sign. This does not always have to be done, because the assembler assumes, if the operand is not prefixed with any other special sign, that extended addressing is to be used. With extended addressing the operand field contains the address of the data to be manipulated. A statement can appear as:

```
STB  >$0445
```

This instruction will store the contents of the B register in location $0445. This machine instruction would be organized in memory at address 1000 as:

```
1000   F7 04 45
```

$F7 is the op code of STB with extended addressing. This is followed by two bytes that contain the effective address where the upper, or most significant, byte of the addresss immediately follows the op code, and the lower, or least significant, byte of the address is last. An instruction using extended addressing can access any memory address from 0 - FFFF (64K) with its two-byte operand address. Machine instructions using extended addressing are made up of the op code (one or two bytes) and the operand address (two bytes) so the instruction will be three or four bytes long. Other examples are: LDU  $2123 and DEC  COUNT. Extended addressing is implied because no other addressing mode is specified. The LDU instruction will load the upper half (bits 15-8) of the U register with the byte read from address $2123 and load the lower half (bits 7-0) of the U register with the byte read from address $2124. The DEC instruction will decrement the value stored at COUNT.

**EXTENDED INDIRECT ADDRESSING**

This addressing mode adds another level of capability to the MC6809E instructions. The effective address when using extended indirect addressing is contained in the two bytes pointed to by the operand field. Essentially, the operand field points to the address of the operand. Thus, the effective address, that is, the address to be finally used, is found indirectly. An example of a statement specifying extended indirect addressing is:

```
STA  [$1022]
```

The bracketed operand indicates the effective address is to be found indirectly. Before this instruction is executed the effective address should have been stored starting at address $1022. Suppose $31F2 was stored at locations $1022 and $1023.  The above STA instruction could be executed and result in the contents of the A register stored in address $31F2. As you can see, the operand field points to where the effective address is stored.

At the machine instruction level the op code will specify indexed addressing, as this is a special case of indexed addressing. A postbyte will follow to specify extended indirect addressing. See Table 4-1. The postbyte will contain $9F to specify indirect addressing. Following the postbyte is the two-byte operand field. The STA instruction would appear in memory at address 1000 as:

```
1000   A7 9F 10 22
1022   31 F2
```

$A7 is the STA with indexed addressing op code. The effective address would be in memory at address $1022.

## REGISTER ADDRESSING

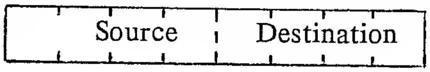
The Exchange, Transfer, and the stack push and pull instructions use register addressing to specify which registers will be acted upon. The operand field of a statement representing one of these instructions would contain the registers to be used. Examples are:

```
TFR  A,B
EXG  D,X
```

For example, the Transfer (TFR) instruction will cause the contents of the A register to be transfered to the B register. The Exchange (EXG) instruction will cause the contents of the D and X registers to be exchanged. In these examples a register pair is in the operand field, but at the machine instruction level the register pair is contained in a postbyte.

In the EXG and TFR instructions the most significant nibble (bits 7-4) of the postbyte contains the code of the source register and the least significant nibble (bits 3-0) contains the code of the destination register. The source and destination nomenclature apply mainly to the TFR instruction, where the contents of the source register is transfered to the destination register. The format of the postbyte and register codes can be seen in Fig. 4-1.

Postbyte



Register codes:

| | |
|---|---|
| 0000 = D | 0101 = PC |
| 0001 = X | 1000 = A |
| 0010 = Y | 1001 = B |
| 0011 = U | 1010 = CCR |
| 0100 = S | 1011 = DPR |

Fig. 4-1 The Transfer/Exchange Postbyte
Courtesy of Motorola, Inc.

The example TFR and EXG machine instructions would be organized in memory starting at address 3000 as:

```
3000    1F 89
3002    1E 01
```

Each instruction is two bytes long. At address 3000 is $1F, the op code of the TFR instruction, followed by $89, its postbyte. At address 3002 is $1E, the EXG op code, followed by $01, its postbyte.

The stack push and pull instructions are PSHS, PSHU, PULU, and PULS. They are all capable of pushing registers on or pulling registers from a stack in a fixed order. The order in which registers are pushed on or pulled from a stack is determined by the postbyte. The registers to be pushed or pulled are designated in the operand field of the instruction statement. For example: PSHU  A,X,Y  will push the contents of the Y, X, and A registers onto the U stack in that order.

The registers to be pushed or pulled are indicated in the machine instruction by a postbyte. Each bit of the postbyte corresponds to one of the eight registers that can be pushed or pulled. The format of the postbyte can be seen in Fig. 4-2. If a register's corresponding indicator bit is set, it will be pushed or pulled. When the MPU is pushing registers on a stack, the order is determined by scanning the postbyte from left to right, and if a bit is set, the corresponding register is pushed on the stack. In the case where all the registers will be pushed, the PC register is the first and the CC register is the last to be pushed. When the MPU is pulling registers from a stack, the order is determined by scanning the postbyte from right to left, and if a bit is set, the corresponding register is pulled. Where all the registers will be pulled, the first register to be pulled is the CC register and the last is the PC register. This technique is used to ensure that the contents of the various registers are not mixed. The order is also exactly the same order as when an interrupt or interrupt return causes the registers to be pushed on or pulled from the S stack.

```
        Postbyte

 | 7  6  5  4  3  2  1  0 |
     push order ->
     pull order <-
```

| | |
|---|---|
| bit 0 - CCR | bit 4 - X |
| bit 1 - A | bit 5 - Y |
| bit 2 - B | bit 6 - S/U |
| bit 3 - DPR | bit 7 - PC |

Fig. 4-2 The Push/Pull Postbyte (Courtesy of Motorola, Inc.)

The only bit of the postbyte that needs further explanation is bit 6. If this bit is set the U register will be pushed on (PSHS) or pulled from (PULS) the S stack, or the S register will be pushed on (PSHU) or pulled from (PULU) the U stack.  Essentially, the stack pointer can not be pushed on or pulled from its own stack.

The PSHU A,X,Y instruction in machine language would appear in memory at address 1000 as:

```
1000   36 32
```

$36 is the PSHU op code and $32 is the postbyte directing that the Y, X, and A registers are to be pushed on the U stack, in that order. The contents of these registers would be stored in memory on the U stack, as depicted in Fig. 4-3.

| Address | Contents | |
|---|---|---|
| U-5 | A | new top of stack |
| U-4 | $X_U$ | |
| U-3 | $X_L$ | |
| U-2 | $Y_U$ | |
| U-1 | $Y_L$ | |
| U | ▭ | old top of stack |

Fig. 4-3 The Y, X, and A Registers Stored in the U Stack

## INDEXED ADDRESSING

The indexed addressing mode of the MC6809E is the most powerful and flexible means of accessing memory locations. In fact, the indexed addressing mode can be divided into four submodes. In all the submodes an index register, either X, Y, U, or S, will be specified in the operand field of a statement and used to calculate the effective address. The indexed addressing mode also makes the most extensive use of the postbyte.

### Zero Offset Indexed Addressing

The simplest submode of indexed addressing is zero offset indexed addressing. The index register in the operand field of a statement contains the address of the data to be accessed by that instruction. In the following example:

```
LDY  #$1908
STA  ,Y
```

The Y index register is first loaded with an address, or pointer, of $1908. Then the Store A (STA) instruction stores the contents of the A register at the address contained in the Y register. The STA instruction statement can also be in the form of STA 0,Y. The effective address is the contents of the Y register plus zero. Another example is:

```
LDU  #$2200
LDX  ,U
```

The U register points to the data at address $2200. The Load X (LDX) instruction will load the upper half of the X register with the byte read from address $2200 and the lower half with the byte read from address $2201.

At the machine instruction level, zero offset indexed addressing is implemented with an op code specifying indexed addressing and an appropriate postbyte. The postbyte formats of the various submodes of indexed addressing can be found in Table 4-1.

| TYPE | FORMS | ASSEMBLER FORM | POST BYTE |
|---|---|---|---|
| Constant Offset From R (signed binary) | Zero Offset | ,R | 1RR00100 |
| | 5-Bit Offset | n,R | 0RRnnnnn |
| | 8-Bit Offset | n,R | 1RR01000 |
| | 16-Bit Offset | n,R | 1RR01001 |
| Accumulator Offset From R (signed binary) | A Register Offset | A,R | 1RR00110 |
| | B Register Offset | B,R | 1RR00101 |
| | D Register Offset | D,R | 1RR01011 |
| Autoincrement/ Autodecrement R | Increment By 1 | ,R+ | 1RR00000 |
| | Increment By 2 | ,R++ | 1RR00001 |
| | Decrement By 1 | ,-R | 1RR00010 |
| | Decrement By 2 | ,--R | 1RR00011 |
| Extended Indirect | 16-bit address | [n] | 10011111 |

where R = X, Y, U, or S RR code: 00 - X 10 - U
01 - Y 11 - S

Table 4-1 The Postbytes of the Indexed Addressing Mode
Courtesy of Motorola, Inc.

The postbyte of the STA and LDX examples can be constructed using Table 4-1. The only bits of the postbyte that can be varied when using zero offset are bits 5 and 6, which indicate the index register to use to obtain the effective address. Now the machine instructions can be assembled in memory at address 2000 as:

```
2000   A7 A4
2002   AE C4
```

The op codes were obtained from Appendix B. At address 2000 is the STA with indexed addressing op code, followed by the zero offset postbyte specifying index register Y. At 2002 is the LDX with indexed addressing op code, followed by the zero offset postbyte specifying index register U.

## Constant Offset Indexed Addressing

The next submode is constant offset indexed addressing. In this case the effective address is the sum of the contents of the index register and a constant number, the offset. The offset is represented with signed binary. In the process of calculating the effective address, the contents of the index register will not be changed. This submode can be further divided into three forms, and each form will use an offset composed of a different number of bits.

The first form uses an offset represented in five bits containing a signed binary number. This form will be used if the offset is in the range of -16 - +15 decimal and not equal to zero. You can check to see that a five-bit signed binary number has the same range. Two statements that specfy this form are:

```
LDA  7,S
STB  -3,X
```

Each of the index registers used was previously loaded with an appropriate value, such as the starting address of a table. You should recognize that the offsets here are decimal, since they are not prefixed with $. The LDA instruction will load the A register with the byte read from the effective address, calculated as the contents of the S register plus 7. The STB instruction stores the contents of the B register in the effective address, calculated as the sum of the contents of the X register plus -3. After executing these instructions, neither the S nor X registers' contents will have changed.

The machine instructions for these examples will be composed of the op code and a postbyte, where the five-bit offset field is contained within the postbyte. The postbyte format can be determined from Table 4-1. The RR bits of the postbyte specify the index register to use to calculate the effective address. The nnnnn bits are the five-bit field which contains the signed binary offset. Therefore, the two instructions will appear in memory at address 1000 as:

```
1000   A6 67
1002   E7 1D
```

The LDA instruction is at address 1000 and the STB instruction is at address 1002.

The next form of constant offset indexed addressing uses an eight-bit field to contain the signed binary equivalent of the offset. This form will be invoked when the offset ranges from decimal -128 - +127 and is outside the range of decimal -16 - +15. Two examples are:

```
INC  $3E,U
STD  -53,X
```

The U and X registers were previously loaded with appropriate addresses. In the first example the INC instruction will increment the contents of the effective address by one. The address is calculated as the contents of the U register plus $3E. The STD instruction will store the contents of the upper half of the D register at the effective address and the lower half at the effective address plus one. The effective address is the sum of the contents of the X register and decimal -53.

The machine instruction is composed of the op code, postbyte, and an offset byte residing in that order in consecutive memory locations. The op code can be found in Appendix B and the format of the eight-bit offset postbyte can be found in Table 4-1. The offset byte will contain just the signed binary representation of the offset. Therefore, the two examples would appear in memory at address as:

```
1000   6C 3E 60
1003   ED 88 CB
```

Each instruction is three bytes long. The INC instruction is at address 1000 and the STD instruction is at address 1003.

The last form of constant offset indexed addressing uses a 16-bit (two-byte) field to contain the signed binary equivalent of the offset. This form is used if the offset ranges from decimal -32768 - +32767 and is outside the range of decimal -128 - +127. An example is COM 1725,S. The S register was previously loaded with an appropriate address. This Complement (COM) instruction will complement all the bits stored at the effective address. In this example the effective address is the sum of the contents of the S register and decimal 1725.

The machine instruction exists as an op code, post byte, and two bytes containing the signed binary offset. The op code can be found in Appendix B and the 16-bit offset post byte format can be found in Table 4-1. The instruction would be organized in memory at address 1000 as:

```
1000   63 E9 06 BD
```

The $63 is the COM with indexing op code and $E9 is the post byte. $06BD is the signed binary equivalent of the offset, 1725.

**Accumulator Offset Indexed Addressing**

The accumulator offset indexed addressing submode adds the contents of an accumulator (A, B, or D register) to the contents of the index register to calculate the effective address. The offset value in the accumulator is in signed binary so the offset can be either negative or positive. An example is:

```
ADDB  #$04
LDA   B,Y
```

The Y register was previously loaded with an appropriate value. The ADDB instruction will add the immediate value of 4 to the contents of the B register, or increment the B register by 4. Then the LDA instruction loads the A register with the byte read from the effective address of the sum of the contents of the Y register and the contents of the B register. Here is one of the advantages of this addressing mode; the offset value can be calculated by the program. The accumulator and index registers are not changed by calculating the effective address.

The machine instruction is composed of the op code and an appropriate postbyte determined from Table 4-1. The above LDA instruction would appear in memory at address 3400 as:

```
3400  A6 A5
```

$A6 is the op code of the LDA with indexed addressing, and $A5 is the postbyte.

**Auto-increment/Auto-decrement**

The auto-increment and auto-decrement forms of indexed addressing allow automatic incrementing or decrementing of the index register to easily access sequential memory locations. These addressing forms result in a modified index register after an instruction is executed.

An instruction with auto-incrementing specified will perform its operation using the effective address contained in the index register. Then the index register is incremented by one or two, as specified. Auto-incrementing by one is specified by a plus (+) sign after the index register in the operand field. Auto-incrementing by two is specified by two plus signs. Two examples are:

```
LDX  #$2400
STA  ,X+
STU  ,X++
```

The X register is first loaded, by the LDX instruction, with an address of $2400. The STA instruction stores the contents of the A register at the address ($2400) contained in the X register. Then the value in the X register is incremented by one, as specified by the single + sign. The X register now contains a $2401. The STU instruction stores the contents of the upper half of the U register at address $2401 and the contents of the X register is incremented by one. Then the upper half of the U register is stored in address $2402 and the X register is again incremented by one resulting in the X register containing $2403. As you can see, this relieves us of having to increment a pointer or offset to point to the next item in a table.

An instruction with auto-decrementing specified will first decrement the contents of the index register by one or two and then proceed with the instruction's execution using the new effective address contained in the index register. Auto-decrementing by one is specified by a minus (-) sign before the index register in the operand field. Auto-decrementing by two is specified by two minus signs. Two examples are:

```
LDU  #$3110
STB  ,-U
LDX  ,--U
```

The LDU instruction loads the U register with the address $3110. The STB instruction starts by decrementing the U register by one, as specified by the single minus sign, so the U register will contain $310F. Then the contents of the B register are stored at address $310F. The LDX instruction will first decrement the U register by two, as specified by the two minus signs; resulting in the U register containing $310D. Then the bytes read from $310D and $310E are loaded into the respective upper and lower halves of the X register.

The auto-incrementing and auto-decrementing of index registers were designed to operate in the same fashion as a stack pointer so this addressing mode can be used when accessing a stack. When pulling or pushing a register off or on a stack, the increment or decrement value should be one for a one-byte register and two for a two-byte register. For example: LDA ,S+ will pull a byte off the S stack into the A register and increment the S register so it points to the next lower byte in the stack.

At the machine instruction level this LDA instruction would appear in memory at address 1000 as:

```
1000   A6 E0
```

$A6 is the op code of the LDA with indexed addressing. The $E0 postbyte specifies auto-incrementing by one of the S index register.

## INDEXED INDIRECT ADDRESSING

In indexed indirect addressing the operand field indirectly specifies the effective address. This is similar to extended indirect addressing, in which the operand field points to where the effective address is stored. All the indexed addressing modes, except auto-increment/auto-decrement by one and 5-bit offset may be used in indexed indirect addressing. If an instruction statement were to appear to specify 5-bit offset indexed indirect addressing, the assembler will interpret it as an 8-bit offset and assemble the machine instruction accordingly. Indexed indirect addressing is specified in a statement by enclosing the operand in brackets, [ ]. Examples of all legal addressing modes are:

```
LDB  [,Y]
STA  [12,X]
LDX  [B,Y]
STX  [,--S]
LDD  [,U++]
```

Each operand points to a memory location where the effective address is stored. That effective address is read from memory by the MPU and used to perform that instruction.

A detailed example of indexed indirect addressing follows. Memory locations $2310 and $2311 contain the address $1A0A. The X register contains $2300. The instruction: STA  $10,X  will form an address of 2310 by adding the offset, $10, to the contents of the X register. Then the effective address will be read out of addresses $2310 and $2311, yielding a $1A0A. Finally the contents of the A register are stored at address $1A0A.

The machine instruction is composed of an op code specifying indexed addressing, a postbyte specifying indirect addressing, and an offset of one or two bytes. The formats of the postbytes of the various types of indexed indirect addressing can be found in Table 4-2.

| TYPE | FORM | ASSEMBLER FORM | POST BYTE |
|---|---|---|---|
| Constant Offset From R | Zero Offset | [,R] | 1RR10100 |
| | 8-Bit Offset | [n,R] | 1RR11000 |
| | 16-Bit Offset | [n,R] | 1RR11001 |
| Accumulator Offset From R | A Register Offset | [A,R] | 1RR10110 |
| | B Register Offset | [B,R] | 1RR10101 |
| | D Register Offset | [D,R] | 1RR11011 |
| Auto-increment/ Auto-decrement | Increment by 2 | [,R++] | 1RR10001 |
| | Decrement by 2 | [,--R] | 1RR10011 |

where R = X, Y, U, or S          RR code: 00 - X   10 - U
                                          01 - Y   11 - S

Table 4-2 The Postbytes of Indexed Indirect Addressing Modes
Courtesy of Motorola, Inc.

The STA example instruction would appear in memory at address 1000 as:

```
1000    A7 98 10
```

The indexed addressing mode is 8-bit offset. $A7 is the STA with indexed addressing op code obtained from Appendix B. The $98 is the indirect 8-bit offset postbyte obtained from Table 4-2. The $10 is the offset byte.

## RELATIVE ADDRESSING

Relative addressing is used only with branch instructions, where the address to branch to is relative to the PC register's contents. A branch instruction's operation is to see if certain CC register bit states match the condition it's looking for. If they match, the PC register's contents are modified, forcing the MPU to execute instructions at some other address. Remember, the PC register contains the address of the next instruction to be performed.

Relative addressing is specified in a statement whenever the mnemonic is that of a branch instruction. This is actually the PC relative addressing mode; therefore all the branch instructions are position independent. Two types of branches are the long branch and the short branch. Each is available for every condition of the CC register bits that can be tested for a match. For example, the Branch on Carry Set (BCS) will cause a branch if the carry bit of the CC register is set. Its short and long forms are specified by the mnemonics BCS and LBCS, respectively.

The short branch machine instruction exists in memory as a single byte op code followed by an offset byte containing the signed binary representation of the offset. If the branch is taken, the offset byte is added to the PC register and the MPU starts to execute instructions at the new address in the PC register. Since the offset is signed binary and eight bits long, the short branch can branch forward only 127 memory addresses, or backward 128 addresses. An example of a BCS instruction at address 1000 with an offset of decimal +36 is:

```
1000  25 24
```

The $25 is the BCS op code from Appendix B and the $24 is the offset byte representing a decimal +36. After fetching this instruction, the PC register will contain $1002, the address of the next instruction. If the carry bit is set, the branch operation will be performed by adding the offset byte, $24, to the PC register, $1002, with the result that the PC register will contain $1026. The MPU then fetches and starts executing the instruction at address $1026.

The long branch instruction exists as an op code two bytes long and a two-byte offset. In this case the offset value can range from decimal -32768 - +32767. Essentially, the long branches perform the same as the short branches, except they can branch further forward and backward. The short branch consumes fewer memory locations and executes faster.

When composing assembly language statements, normally one is not much concerned about relative addressing since the assembler will calculate the offset values. To decide which branch to use, use the short form because of its speed and memory savings. A branch beyond the limits of 127 forward or 128 backward will display a **byte overflow** error message. You can then change just the required branches to the long form.

## DIRECT ADDRESSING

In direct addressing the op code is followed by one byte, the lower byte of the effective address. The upper byte is in the direct page (DP) register. During a RESET operation, the DP register is cleared to 00. To use this addressing mode the DP register must be loaded with the appropriate value. Unfortunately, there is no Load DP instruction; it is loaded as follows:

```
LDA  #$20
TFR  A,DP
```

The LDA instruction loads the A register with the desired value, $20 in this example. Then the TFR instruction transfers the contents of the A register into the DP register.

The direct addressing mode is invoked by prefixing the operand with a < sign. Two examples are:

```
LDA  <$DE
STU  <$2082
```

The DP register was previously loaded with $20. The LDA instruction's effective address is 20DE, the combination of the DP register and the operand. The assembler will assemble the STU instruction as STU <$82. It will drop the most significant byte of a two-byte operand if we have informed the assembler of the DP register's contents with SETDP command (described in Chapter Six) and the most significant byte of the operand address is equal to the contents of the DP register.

The machine instructions will exist as op codes specifying direct addressing, followed by the lower address byte. They would appear at address 1000 as:

```
1000   96 DE
1002   DF 82
```

The LDA with direct addressing op code is the $96 at address 1000, followed by the lower address byte of $DE. The STU with direct addressing op code is the $DF at address 1002, followed by the lower address byte of $82.

The advantages of direct addressing are that each instruction takes fewer memory locations and they execute faster. The disadvantages are that the DP register must be loaded with an appropriate value. Also, the range of effective addresses is 256, the range of the lower address byte. If you want to access a location outside this range, the DP register must be loaded with a new value.

## PROGRAM COUNTER RELATIVE ADDRESSING

Program counter, or PC, relative addressing is a type of indexed addressing wherein the PC register is used as an index register. It allows one to easily write a program that is position independent if the instructions accessing memory locations within that program use this addressing mode. A program that is position independent can be located at almost any series of addresses and still run correctly.

Before explaining this addressing mode another assembly language concept needs to be introduced. The symbol field of a statement is a group of character positions in which a symbol, name, or label can be put. Symbol, label, and name are equivalent terms and represent the address of the item in the command and operand fields of that statement. The label relieves us of having to work with absolute addresses within a program.

**PC Relative Addressing**

The PC relative addressing mode is specified in a statement by appending ,PCR to the operand. This can be seen as:

```
                STA   TABLE,PCR
                LDX   TABLE
                   .

                   .
        TABLE    RMB   20
```

The STA statement is using PC relative addressing. The bottom statement uses the mnemonic of RMB, a command to tell the assembler to reserve memory bytes, in this case 20 decimal. The reserved 20 sequential memory locations is named TABLE, and the assembler has assigned the starting address of those 20 bytes to be equivalent to TABLE. The assembler will construct the LDX instruction using the TABLE address. In the case of the STA instruction, the offset of TABLE from what the PC register would contain when executing the STA instruction is calculated and used in assembling the STA with PC relative addressing machine instruction. The STA instruction will store the contents of the A register at TABLE, the starting address of the series of 20 reserved bytes.

If this program were put in some other memory addresses, the STA instruction would still access the memory location a fixed number of bytes offset from it; in this case, the new location of TABLE. The LDX instruction, which does not use PC relative addressing, would access the original address of TABLE and not its new address. Thus, the STA is relocatable and the LDX is not.

The machine instruction exists in one of two forms, depending on whether the offset will fit in an 8 or 16-bit offset field using signed binary representation. EDTASM+, will use the 16-bit offset field whenever a labeled operand is detected. The op code will be an op code specifying indexed addressing, as this is a type of indexed addressing. The postbyte can be found in Table 4-3.

| TYPE | FORM | ASSEMBLER FORM | POST BYTE |
|---|---|---|---|
| Constant Offset From PC | 8-Bit Offset<br>16-Bit Offset | n,PCR<br>n,PCR | 1xx01100<br>1xx01101 |

where xx = don't care

Table 4-3 The Postbytes of PC Relative Addressing
Courtesy of Motorola, Inc.

Following the postbyte will be one or two bytes containing the offset. The effective address will be the sum of the offset and the contents of the PC register. Note that the effective address is calculated relative to the location of the next instruction, since the PC register has been incremented to point to it. An example of a STA instruction with a PC relative offset of a decimal +25 is: A7 8C 19. $A7 is the STA with indexed addressing op code and $8C is the postbyte, indicating an 8-bit offset. The offset byte contains $19, the signed binary equivalent of decimal +25.

If an absolute address outside the program is accessed, the position independency is lost. Two examples are:

```
STA  $1700,PCR
LDX  1000,PCR
```

In the first example the STA instruction will store the contents of the A register in address $1700. In the second example the X register will be loaded with the two bytes read from the decimal addresses of 1000 and 1001. The assembler constructs the instructions at some address in memory, and the offset to the absolute address will be calculated and put in the offset byte(s). If we later moved those instructions to another address, they would not access the desired memory locations because their offsets would not have changed, but the contents of the PC register would have.

**Indirect PC Relative Addressing**

PC relative addressing can also be used to indirectly specify the effective address. This is called indirect PC relative addressing. In this addressing mode the operand field points to where the effective address is stored in memory. This addressing mode is specified in a statement by enclosing the operand in brackets as seen below.

```
STB  [TABLE,PCR]
```

In this case the effective address is located in the first two bytes, starting at TABLE. The MPU will read the effective address from memory and use it to direct where the contents of the B register will be stored.

The machine instruction exists as an op code specifying indexed addressing followed by a postbyte, and then a one or two-byte offset. The postbyte will indicate indirect PC relative addressing and whether the offset is contained in one or two bytes. The postbyte formats can be found in Table 4-4. The above instruction with a two-byte offset of a decimal -202 would appear in memory as: E7 9D FF 36. $E7 is the STB with indexed addressing op code followed by the $9D postbyte, as found in Table 4-4. The $FF36 is the signed binary offset representation of decimal -202.

| TYPE | FORM | ASSEMBLER FORM | POST BYTE |
|---|---|---|---|
| Constant Offset From PC | 8-Bit Offset<br>16-Bit Offset | [n,PCR]<br>[n,PCR] | 1xx11100<br>1xx11101 |

where xx = don't care

Table 4-4 The Postbytes of Indirect PC Relative Addressing
Coutesy of Motorola, Inc.

This completes the descriptions of the various addressing modes. If you are already familiar with assembly language and have an assembler, try writing some programs.

# CHAPTER 5

# MC6809E Instruction Set

The 59 instructions of the MC6809E are its instruction set. This chapter describes the operation of the instructions; some very similar instructions may be grouped together in one description. The descriptions are titled with the instruction mnemonics and are in alphabetical order. Within each description is the instruction's source form, type, operation, affect on the CC register bits, legal addressing modes, and the description of its operation, with examples.

The **source form** of an instruction is the manner in which the mnemonic and any operand would appear in a statement. A list of statements representing a program is known as the **source code**. An assembler reads the source code and generates the machine instructions corresponding to each statement; the resulting machine instructions are known as the **object code**.

Instructions can be classified by type: data movement, arithmetic, logic, test, branch, and miscellaneous. Data movement instructions are those that load, store, transfer, or exchange data. Also included are instructions that push or pull data to or from a stack. Arithmetic instructions are those that add, subtract, increment, decrement, clear a byte, negate a byte, operate on BCD numbers, multiply, and perform sign extension. Logic instructions perform the logical operations of AND, OR, EOR, shift or rotate a byte, complement, and AND or OR immediate data with the CC register. Test instructions can perform an arithmetic or logical comparison of data in a register with data in memory, or test for a zero, minus, or positive value in a memory location or the A or B register. Branch instructions let the MPU make decisions based on the conditions of certain bits in the CC register. These instructions should be used immediately after an instruction that modifies the CC register, if a

| Symbol | Description |
|---|---|
| A | The A register. |
| B | The B register. |
| CC | The condition code register. |
| D | The D register. |
| DD | An 8-bit offset of a branch instruction. |
| DDDD | A 16-bit offset of a branch instruction. |
| DP | The direct page register. |
| EA | The effective address. |
| LSN | Least significant nibble. |
| MSN | Most significant nibble. |
| M | Memory location contents specified by the operand field. |
| MI | The immediate byte following an op code |
| P | An operand field specifying a memory location to be accessed by the immediate, direct, extended, indexed, or the special cases of indexed addressing. |
| PC | Program counter register. |
| Q | An operand field specifying a memory location by the direct, extended, indexed, or the special cases of indexed addressing. A byte will be read, modified, and written back to it. |
| R | A register before an operation. |
| R' | A register after an operation. |
| S | The S register. |
| TEMP | A register within the MPU where results are temporarily stored. |
| U | The U register. |
| X | The X register. |
| xx | A byte (two hexadecimal digits). |
| $xx_U$ | Most significant byte of a 16-bit number. |
| $xx_L$ | Least significant byte of a 16-bit number. |
| Y | The Y register. |
| ( ) | Contents of a register in parentheses is used as an address. |
| # | Indicates the immediate addressing mode. |
| $ | Indicates the following number is hexadecimal |
| [ ] | Brackets around the operand indicate indirect addressing. |
| , | In operand field, indicates indexed addressing. |
| <- | Indicates a transfer of data. |
| ∧ | Logical AND. |
| ∨ | Logical OR. |
| ⊻ | Logical exclusive OR. |
| ‾ | Logical NOT or complement. |
| : | Indicates the quantities are concatenated. |
| +,-,x | Arithmetic addition, subtraction, or multiplication. |

Table 5-1 Abbreviations and Meanings.

decision based on the results of that instruction is desired. The miscellaneous instructions are those that do not fit in any of the other types; they include software interrupts, return from interrupt, sync, etc. In the descriptions, abbreviations and symbols will be used, whose definitions can be found in Table 5-1.

The condition code description explains which bits of the CC register are affected, and how they are modified by each instruction. These condition code bits will be set or cleared on completion of the executing instruction, unless stated otherwise. The addressing modes each instruction can use are quite straightforward, except for indexed addressing. In the following descriptions, indexed addressing includes all the indexed addressing submodes plus those special cases technically considered to belong to this mode. The special cases are: extended indirect, indexed indirect, PC relative, and PC relative indirect addressing. The RESET and interrupt operations are also included at the end of this chapter; they perform functions similar to instructions.

## ABX — Add Accumulator B into the X Register

Source Form: ABX
Operation: X' <- X+B — Type: Arithmetic
Addressing Modes: Inherent. — Condition Codes: Not Affected.

Description: The contents of the B register, in straight binary, are added to the contents of the X register; the result is routed back into the X register. For example:

```
LDB #$21
LDX #$1091
ABX
```

LDB and LDX load the immediate values of $21 and $1091, respectively, into the B and X registers. After the ABX instruction executes the X register contains $10B3, the sum of the contents of the B register and the original contents of the X register. Note that the contents of the B register are not changed.

## ADC — Add with Carry into Register

Source Forms: ADCA P ; ADCB P
Operation: R' <- R+M+C — Type: Arithmetic
Addressing Modes: Immediate, Extended, Direct, and Indexed.
Condition Codes:

- H - Set if the operation caused a half carry, or carry from bit 3 in the ALU; cleared otherwise.
- N - Set if bit 7 of the resulting byte is set; cleared otherwise. N set will indicate a negative result if using signed binary.
- Z - Set if the resulting byte is zero; cleared otherwise.

V - Set if an overflow or underflow occurred; cleared otherwise. This applies only to signed binary numbers.
C - Set if a carry out of bit 7 in the ALU is generated by this operation; cleared otherwise.

Description: A byte is read from a memory location as specified by the operand field. The sum of that byte, the current carry (C) bit, and the contents of the A or B register (ADCA or ADCB) is routed back to the A or B reigster. An example of ADCB using extended addressing can be set up. Suppose the B register contains $68, the C bit is clear, and memory location $2820 contains $A5. The instruction ADCB $2820 will execute using the following process:

```
                0        current state of C bit
        1010 0101        byte read from $2820
       +0110 1000        current contents of B
C=1     0000 1101        new contents of B
```

The B register will contain $0D and the C bit will be set. The H, N, Z, and V bits are cleared. This instruction would be used when adding two strings of bytes such as:

```
  XX XX XX XX
+ YY YY YY YY
```

The addition would be performed byte by byte from right to left. Any carry generated by adding of two bytes would be added, by the ADC instruction, with the next two bytes to the left; in effect, propagating any carries.

## ADD (8 Bit) — Add Memory into Register

Source Forms: ADDA P ; ADDB P
Operation: R' <- R+M — Type: Arithmetic
Addressing Modes: Immediate; Extended; Direct; Indexed.
Condition Codes:
H - Set if the operation caused a half carry, or carry from bit 3 in the ALU; cleared otherwise.
N - Set if bit 7 of the resulting byte is set; cleared otherwise. The N bit set indicates a negative result if using signed binary.
Z - Set if the resulting byte is zero; cleared otherwise.
V - Set if an overflow or underflow occurred; cleared otherwise. This applies only to signed binary numbers.
C - Set if a carry out of the ALU, from bit 7, is generated; cleared otherwise.

Description: A byte is read from memory as specified by the operand field. That byte is added to the contents of the A or B (ADDA or ADDB) register and the resulting byte is routed back into the respective A or B register.

If, for example, the A register contains $A3, then the ADDA with immediate addressing: ADDA #$22 will result in the A register containing $C5, the sum of $A3 and $22. This process is shown as:

```
  1 0 1 0  0 0 1 1     current contents of A
+ 0 0 1 0  0 0 1 0     immediate byte from memory
  1 1 0 0  0 1 0 1     new contents of A
```

The CC register bits are also modified so the N bit is set and the H, Z, V, and C bits are cleared.

## ADD (16 Bit) — Add Memory into Register

Source Form: ADDD P
Operation: D' <- D+M:M+1 Type: Arithmetic
Addressing Modes: Immediate; Extended; Direct; Indexed.
Condition Codes:

- H - Not affected.
- N - Set if bit 15 of the resulting 16-bit word is set; cleared otherwise. N set indicates a negative result if using signed binary.
- Z - Set if the resulting word is zero; cleared otherwise.
- V - Set if an overflow or underflow occurred; cleared otherwise. This applies only to signed binary numbers.
- C - Set if a carry is generated from bit 15; cleared otherwise.

Description: Two bytes are read from two consecutive memory locations; the operand field specifies the first address. The two bytes are concatenated to produce a 16-bit word to be added to the contents of the D register. If two bytes are stored at addresses $1900 and $1901 such as:

```
1900  01 7D
```

and the D register contains $6091, the following instruction with extended addressing: ADDD $1900 will result in the D register containing $620E, the sum of $6091 and $017D. The CC register would also be modified so that N, Z, V, and C are clear.

## AND — Logical AND Memory into Register

Source Forms: ANDA P; ANDB P
Operation: R' <- R $\wedge$ M Type: Logic
Addressing Modes: Immediate, Extended, Direct, Indexed.
Condition Codes:

- H - Not affected.
- N - Set if bit 7 of the resulting byte is set; cleared otherwise. The N bit set indicates a negative result if using signed binary.

Z - Set if the resulting byte is zero; cleared otherwise.
V - Always cleared by this instruction.
C - Not affected.

Description: A byte is read from a memory address as specified by the operand field. That byte is ANDed with the contents of the A or B (ANDA or ANDB) register and the resulting byte is routed back to the same A or B register. Here is an example using indexed addressing: a byte of $77 is stored at address $2280 and the B register contains $DF. Then:

```
LDY #$2280
ANDB ,Y
```

will first load (LDY) the Y register with the address of interest. The ANDB instruction will read a byte from the address specified in the Y register and AND it with the contents of the B register. The result, $57, is then routed back into the B register. This can be demonstrated as:

```
   1 1 0 1  1 1 1 1      current contents of B
 ^ 0 1 1 1  0 1 1 1      byte read from memory
   ----------------
   0 1 0 1  0 1 1 1      new contents of B
```

The CC register is also modified so the N, Z, and V bits are cleared. A typical use for this instruction is to clear selective bits in the A or B register. The bits to be cleared are those that are clear in the operand byte. This can be seen above.

## ANDCC — Logical AND Immediate Byte into CC Register

Source Form: ANDCC #xx
Operation: CC' <- CC ∧ MI — Type: Logic
Addressing Modes: Immediate.
Condition Codes: Any or all bits may be affected. See description.

Description: The immediate byte is read from memory and ANDed with the contents of the condition code register. The resulting byte is then routed back into the CC register. For example, if the CC register contains $38, the instruction: ANDCC #$F7 will result in the CC register containing $30. Note that this instruction can be used to clear particular bits in the CC register. In this example, bit 3 of the CC register was cleared since bit 3 of the operand was clear.

## ASL — Arithmetic Shift Left

Source Forms: ASL Q; ASLA; ASLB
Operation: C <- [7 <- 0] <- 0 — Type: Arithmetic
Addressing Modes: Inherent; Extended; Direct; Indexed.

Condition Codes:
- H - Its state is not defined after this instruction. It may change to some unpredicted state, therefore a branch instruction that tests its state should not be used after an ASL instruction.
- N - Set if bit 7 of the resulting byte is set; cleared otherwise.
- Z - Set if the resulting byte is zero; cleared otherwise.
- V - Set if the exclusive OR of bits 7 and 6 of the original operand is 1; cleared otherwise.
- C - Set if bit 7 of the original operand was set; cleared otherwise.

Description: The bits of a byte are shifted one bit position to the left, depicted by the above operation, where the byte may be a byte read from memory (ASL Q) or the contents of the A or B (ASLA or ASLB) register. In the shift left process, bit 0 of the resulting byte is cleared and the original state of bit 7 is shifted into the C bit of the CC register. The resulting byte is then returned to its source - memory or the A or B register. For example, suppose $55 is stored at address $18E0. The ASL with extended addressing: ASL $18E0 will read the byte, $55, from address $18E0. The shift operation is performed as:

```
        0 1 0 1 0 1 0 1        original byte
C <-    0 1 0 1 0 1 0 1 <- 0   shift
C=0     1 0 1 0 1 0 1 0        resultant byte
```

The resulting byte of $AA is stored back in memory address $18E0. Also, the CC register bits are modified; the C and Z bits are cleared, the N and V bits are set, and the H bit may be either set or clear.

## ASR — Arithmetic Shift Right

Source Forms: ASR Q; ASRA; ASRB

Operation: | 7 -> 0 | -> C    Type: Arithmetic

Addressing Modes: Inherent; Direct; Indexed; Extended.

Condition Codes:
- H - Its state is not defined after this operation, therefore a branch instruction testing its state should not follow this instruction.
- N - Set if bit 7 of the resulting byte is set; cleared otherwise. N set indicates a negative result if using signed binary numbers.
- Z - Set if the resulting byte is zero; cleared otherwise.
- V - Not affected.
- C - Set if bit 0 of the original operand was set; cleared otherwise.

Description: A byte's bits are shifted one bit position to the right as depicted in the above operation; the byte may have been read from memory (ASR Q) or come from the A or B (ASRA or ASRB) register. In the right shift process bit 7 is unchanged and bit 0 is shifted out of the byte into the C bit. The resulting byte is returned to its source - memory or the A or B register. For example, suppose the B register contains $85. The ASR with inherent addressing: ASRB will perform the arithmetic shift right operation on the contents of the B register as:

```
         1 0 0 0 0 1 0 1        original contents of B
         1 0 0 0 0 1 0 1 -> C  shift
    C=1  1 1 0 0 0 0 1 0        result
```

The resulting byte of $C2 is now routed back to the B register. The CC register is modified where the C and N bits are set, the Z bit is cleared, and the H bit may be set or clear.

## BCC — Branch on Carry Clear

Source Forms: BCC DD; LBCC DDDD
Operation: TEMP <- MI                                    Type: Branch
            If C=0, then PC' <- PC+TEMP
Addressing Modes: Relative.          Condition Codes: Not affected.

Description: The BCC will perform a branch if the C bit is clear; otherwise processing will continue with the instruction following the BCC. The BCC is equivalent to the BHS instruction.

## BCS — Branch on Carry Set

Source Forms: BCS DD; LBCS DDDD
Operation: TEMP <- MI                                    Type: Branch
            If C=1, then PC' <- PC+TEMP
Addressing Modes: Relative.          Condition Codes: Not affected.

Description: The BCS will perform a branch if the C bit is set; otherwise processing will continue with the instruction following the BCS. The BCS is equivalent to the BLO instruction.

## BEQ — Branch on Equal

Source Forms: BEQ DD; LBEQ DDDD
Operation: TEMP <- MI                                    Type: Branch
            If Z=1, then PC' <- PC+TEMP
Addressing Modes: Relative.          Condition Codes: Not affected.

Description: The BEQ will perform a branch if the Z bit is set; otherwise processing continues with the instruction following the BEQ. The BEQ is typically used after a compare instruction and will cause a branch if the two compared quantities were exactly the same. Used after a subtract instruction, it will branch if the result is zero.

## BGE — Branch on Greater than or Equal to Zero

Source Forms: BGE DD; LBGE DDDD

Operation: TEMP <- MI Type: Branch

If $(N \veebar V)=0$, then PC' <- PC+TEMP

Addressing Modes: Relative. Condition Codes: Not affected.

Description: The BGE will perform a branch if the N and V bits are both set or both clear; otherwise processing continues with the instruction following the BGE. The BGE is typically used after a subtract or compare of signed binary quantities. After a compare it will branch if the register contents were greater than or equal to the memory operand. After a subtract it will branch if the result is greater than or equal to zero.

## BGT — Branch on Greater Than

Source Forms: BGT DD; LBGT DDDD

Operation: TEMP <- MI Type: Branch

If $Z \vee (N \veebar V)=0$, then PC' <- PC+TEMP

Addressing Modes: Relative. Condition Codes: Not affected.

Description: The BGT will perform a branch if the Z bit is clear and the N and V bits are both set or both clear; otherwise processing will continue with the instruction following the BGT. The BGT is typically used after a compare or subtract of signed binary quantities. After a compare it will cause a branch if the register contents were greater than the memory operand. After a subtract it will branch if the result is greater than zero.

## BHI — Branch if Higher

Source Forms: BHI DD; LBHI DDDD

Operation: TEMP <- MI Type: Branch

If $(C \vee Z)=0$, then PC' <- PC+TEMP

Addressing Modes: Relative. Condition Codes: Not affected.

Description: The BHI will perform a branch if the C and Z bits are both clear; otherwise processing will continue with the instruction following the BHI. The BHI is typically used after a compare or subtract of straight binary quantities. After a compare it will branch if the register contents were greater, or higher, than the memory operand. After a subtract it will branch if the result is greater than zero.

## BHS — Branch if Higher or Same

Source Forms: BHS DD; LBHS DDDD

Operation: TEMP <- MI Type: Branch

If C=0, then PC' <- PC+TEMP

Addressing Modes: Relative. Condition Codes: Not affected.

Description: The BHS will perform a branch if the C bit is clear; otherwise processing will continue with the instruction following the BHS. The BHS is typically used after a subtract or compare of straight binary quantities. After a compare, it will branch if the register contents were higher than or the same as the memory operand. After a subtract, it will branch if the result is zero or greater than zero. The BHS is a duplicate of the BCC, and can be specified by one of two mnemonics, BHS or BCC. This assists the programmer by giving the instructions names conveying the operation of each instruction.

## BIT — Bit Test

Source Forms: BITA P; BITB P

Operation: TEMP <- R ∧ M  Type: Test

Addressing Modes: Immediate; Direct; Extended; Indexed.

Condition Codes:

- H - Not affected.
- N - Set if bit 7 of the resulting byte is set; cleared otherwise. N set indicates a negative number if using signed binary numbers.
- Z - Set if the resulting byte is zero; cleared otherwise.
- V - Always cleared.
- C - Not affected.

Description: The BIT instruction causes a byte to be read from memory, specified by the operand field. That byte is ANDed with the contents of the A or B (BITA or BITB) register and the resulting byte is temporarily stored in the TEMP register, never to be used. The contents of the memory location and the A or B register are not changed; only the CC register is modified. The BIT instruction is used to test the condition of a bit in a memory location. For example, suppose one wanted to know if bit 1 of the byte stored at address $112C is set or clear. This could be found as follows:

```
LDA #$02
BITA $112C
```

LDA loads A with $02 (only bit 1 set). BITA ANDs the contents of address $112C with the contents of A and sets or clears the appropriate CC register bits. If the byte read from memory has bit 1 set, the resulting byte will have only bit 1 set, as seen below:

```
   0000 0010     contents of A
∧  xxxx xx1x     contents of $112C
   ---------
   0000 0010     result
```

The x means we do not care what state that bit is in. The Z bit would be cleared indicating a non-zero value and bit 1 of the operand byte is set.

If bit 1 of the operand was not set, the Z bit would be set, since the result of ANDing would be zero.

## BLE — Branch on Less than or Equal to Zero

Source Forms: BLE DD: LBLE DDDD
Operation: TEMP <- MI  Type: Branch
  If Z $\vee$(N$\veebar$V)=1, then PC' <- PC+TEMP
Addressing Modes: Relative.  Condition Codes: Not affected.

Description: The BLE will perform a branch if the Z bit is set or if either, but not both, the N or V bit is set; otherwise processing will continue with the instruction following the BLE. The BLE can be used after an addition or subtraction of signed binary values where it will branch if the result is less than or equal to zero. It can also be used after a compare of signed binary values where it will branch if the register contents are less than or equal to the memory operand.

## BLO — Branch on Lower

Source Forms: BLO DD; LBLO DDDD
Operation: TEMP <- MI  Type: Branch
  If C=1, then PC' <- PC+TEMP
Addressing Modes: Relative.  Condition Codes: Not affected.

Description: The BLO will perform a branch if the C bit is set; otherwise processing will continue with the instruction following the BLO. The BLO instruction is used after a subtract or compare of straight binary numbers and will branch if the register contents were lower, or smaller, than the memory operand. Note that the BLO is the same as the BCS instruction.

## BLS — Branch on Lower or Same

Source Forms: BLS DD; LBLS DDDD
Operation: TEMP <- MI  Type: Branch
  If (C $\vee$Z)=1 then PC' <- PC+TEMP
Addressing Modes: Relative.  Condition Codes: Not affected.

Description: The BLS will perform a branch if the C or Z bits are set; otherwise processing will continue with the instruction following the BLS. The BLS is used after a subtract or compare of straight binary numbers and will branch if the register contents were lower than or the same as the memory operand.

## BLT — Branch on Less Than Zero

Source Forms: BLT DD; LBLT DDDD
Operation: TEMP <- MI  Type: Branch
  If (N$\veebar$V)=1, then PC' <- PC+TEMP

Addressing Modes: Relative.        Condition Codes: Not affected.

Description: The BLT will perform a branch if either, but not both, the N or V bit are set; otherwise processing continues with the instruction following the BLT. The BLT is used after a subtract or compare of signed binary quantities. After a subtract, it will branch if the result is less than zero. After a compare, it will branch if the register contents are less than the memory operand.

## BMI — Branch on Minus

Source Forms: BMI DD; LBMI DDDD
Operation: TEMP <- MI                                   Type: Branch
            If N=1, then PC' <- PC+TEMP
Addressing Modes: Relative.        Condition Codes: Not affected.

Description: The BMI will perform a branch if the N bit is set; otherwise processing will continue with the instruction following the BMI. Since the BMI does not check for the overflow condition, it is best to use the BLT after an operation on signed binary numbers.

## BNE — Branch on Not Equal

Source Forms: BNE DD; LBNE DDDD
Operation: TEMP <- MI                                   Type: Branch
            If Z=0, then PC' <- PC+TEMP
Addressing Modes: Relative.        Condition Codes: Not affected.

Description: The BNE will perform a branch if the Z bit is clear; otherwise processing will continue will the instruction following the BNE. The BNE, if used after a subtraction, will branch if the result is not zero. If used after a compare, it will branch if the register contents do not equal the memory operand.

## BPL — Branch on Plus

Source Forms: BPL DD; LBPL DDDD
Operation: TEMP <- MI                                   Type: Branch
            If N=0, then PC' <- PC+TEMP
Addressing Modes: Relative.        Condition Codes: Not affected.

Description: The BPL will perform a branch if the N bit is clear; otherwise processing will continue with the instruction following the BPL. The BPL is used after an operation on straight binary quantities to branch if the result is less than $80. If the quantities are signed binary, it is best to use the BGE instruction.

## BRA — Branch Always

Source Forms: BRA DD; LBRA DDDD

Operation: TEMP <- MI — Type: Branch

PC' <- PC+TEMP

Addressing Modes: Relative. — Condition Codes: Not affected.

Description: The BRA will always perform a branch, regardless of the state of the bits of the CC register. The BRA is also known as an **unconditional** branch.

## BRN — Branch Never

Source Forms: BRN DD; LBRN DDDD

Operation: TEMP <- MI — Type: Branch

Addressing Modes: Relative. — Condition Codes: Not affected.

Description: The BRN will never perform a branch; therefore processing always continues with the instruction following the BRN.

## BSR — Branch to Subroutine

Source Forms: BSR DD; LBSR DDDD

Operation: TEMP <- MI — Type: Branch

S' <- S-1, (S) <- $PC_L$

S' <- S-1, (S) <- $PC_U$

PC' <- PC+TEMP

Addressing Modes: Relative. — Condition Codes: Not affected.

Description: The BSR will first push the contents of the PC register into the S stack by decrementing the contents of S by one, and storing the lower half of the PC register at the address now contained in S. Then the contents of S are decremented again by one and the upper half of the PC register contents stored at the new address in S. The BSR instruction always branches to, normally, a subroutine. The last instruction of the subroutine should be an RTS, which will pull the PC register value from the S stack and cause the MPU to return to the program that called the subroutine. There processing will resume with the instruction immediately following the BSR.

## BVC — Branch on Overflow Clear

Source Forms: BVC DD; LBVC DDDD

Operation: TEMP <- MI — Type: Branch

If V=0, then PC' <- PC+TEMP

Addressing Modes: Relative. — Condition Codes: Not affected.

Description: The BVC will perform a branch if the V bit is clear; otherwise processing will continue with the instruction following the BVC. The BVC, used after an operation on signed binary quantities, will branch if no overflow occurred.

## BVS — Branch on Overflow Set

Source Forms: BVS DD; LBVS DDDD

Operation: TEMP <- MI  Type: Branch

If V=1, then PC' <- PC+TEMP

Addressing Modes: Relative.  Condition Codes: Not affected.

Description: The BVS will perform a branch if the V bit is set; otherwise processing will continue with the instruction following the BVS. The BVS is used after an operation on signed binary numbers and will branch if an overflow condition was generated.

## CLR — Clear Accumulator or Memory Location

Source Forms: CLR Q; CLRA; CLRB

Operation: TEMP <- xx  Type: Arithmetic

M <- 00 or R <- 00

Addressing Modes: Inherent; Direct; Indexed; Extended.

Condition Codes:

H - Not afftected.
N - Always cleared.
Z - Always set.
V - Always cleared.
C - Always set.

Description: The contents of the A or B (CLRA or CLRB) register or of a memory location (CLR Q) are cleared so it now contains 00.

## CMP (8 Bit) — Compare Memory from Register

Source Forms: CMPA P; CMPB P

Operation: TEMP <- R-M  Type: Test

Addressing Modes: Immediate; Extended; Direct; Indexed.

Condition Codes:

H - Its state is undefined therefore its state should not be tested with a branch instruction.

N - Set if bit 7 of the byte resulting from the subtraction process is set; cleared otherwise. N set indicates that the contents of M are greater than the contents of the accumulator in use, if comparing signed binary numbers.

Z - Set if the byte resulting from the subtraction process is zero; cleared otherwise. Z set indicates that the a contents of the memory location exactly equal the contents of the accumulator in use.

V - Set if an overflow condition was generated by the subtraction process; cleared otherwise. This bit is applicable only to signed binary numbers.

C - Set if a borrow from bit 7 of the ALU is generated by the subtraction process; cleared otherwise. When comparing straight binary values, C set indicates the accumulator contents are lower than the memory location contents.

Description: The CMP compares the contents of accumulator A or B (CMPA or CMPB) to the contents of a memory location by subtracting the memory contents from the accumulator contents. The resulting byte is not used and the contents of the accumulator and memory location are not changed. Only the bits of the CC register are modified to indicate the result. For example, the A register contains $A3 and memory location $3F22 contains $11. The CMP instruction with extended addressing: CMPA $3F22 will subtract $11 from $A3 clearing the Z, V, and C bits, setting the N bit, and leave the H bit in an undetermined state. The subtraction process is demonstrated as:

```
          b
   1 0 1 0  0 0 1 1     contents of A
 - 0 0 0 1  0 0 0 1     contents of memory
 ------------------
   1 0 0 1  0 0 1 0     unused result
```

## CMP (16 Bit) — Compare Memory from Register

Source Forms: CMPD P; CMPX P; CMPY P; CMPU P; CMPS P

Operation: TEMP <- R-M:M+1 Type: Test

Addressing Modes: Immediate; Extended; Direct; Indexed.

Condition Codes:

H - Not affected.

N - Set if bit 15 of the word resulting from the subtraction process is set; otherwise cleared. N set indicates the memory operand is greater than the register contents if comparing signed binary numbers.

Z - Set if the word resulting from the subtraction process is zero; otherwise cleared. Z set indicates the memory operand is exactly equal to the register contents.

V - Set if the subtraction process resulted in a 16-bit overflow or underflow; cleared otherwise. This applies only to signed binary.

C - Set if a borrow is generated out of bit position 15 during the subtraction process; otherwise cleared. If comparing straight binary numbers, C set indicates the register contents are lower than the memory operand.

Description: The CMP compares the contents of a 16-bit register with the contents of two consecutive memory addresses; the operand field specifies the first address. The compare is performed by subtracting the 16-bit word, read from two memory locations, from the contents of the selected 16-bit register. As a result, the memory locations and the 16-bit register are not modified; only the CC register bits may be set or cleared to indicate the result. For an example of comparing signed binary numbers, the Y register contains $021A and memory locations $1705 and $1706 contain $072E as seen below:

```
1705  07 2E
```

The instruction to compare the Y register with the two-byte memory operand using extended addressing is: CMPY $1705. The compare is performed by reading the two bytes from addresses $1705 and $1706, and concatenating them to form the number 072E. That number is then subtracted from the value in the Y register and the CC register bits are set or cleared accordingly. The subtraction process in hexadecimal is:

```
b  b b b
   0 2 1 A      contents of Y
 - 0 7 2 E      memory operand
 ---------
   F A E C       unused result
```

As a result, the C and N bits are set and the Z and V bits cleared.

## COM — Complement

Source Forms: COM Q; COMA; COMB

Operation: M <- $\overline{xx}$ or R <- $\overline{R}$ Type: Logic

Addressing Modes:Extended; Indexed; Inherent; Direct.

Condition Codes:

H - Not affected.
N - Set if bit 7 of the resulting byte is set; cleared otherwise.
Z - Set if the resulting byte is zero; cleared otherwise.
V - Always cleared.
C - Always set.

Description: The COM instruction will replace the contents of a memory location (COM Q) or the A or B (COMA or COMB) register with its logical complement. The contents will be read, the state of each bit reversed, and then stored where the original was taken. For example, the A register contains $E6. The instruction: COMA results in the A register containing $19. In the CC register, the C bit is set and the N, V, and Z bits are cleared.

## CWAI — Clear CC bits and Wait for Interrupt

Source Forms: CWAI #$xx

Operation: CC' <- CC ∧ MI

Type: Miscellaneous

```
Set E bit
S' <- S-1, (S) <- PC_L
S' <- S-1, (S) <- PC_U
S' <- S-1, (S) <- U_L
S' <- S-1, (S) <- U_U
S' <- S-1, (S) <- Y_L
S' <- S-1, (S) <- Y_U
S' <- S-1, (S) <- X_L
S' <- S-1, (S) <- X_U
S' <- S-1, (S) <- DP
S' <- S-1, (S) <- B
S' <- S-1, (S) <- A
S' <- S-1, (S) <- CC
Wait for interrupt
```

Addressing Modes: Immediate.

Condition Codes: All bits may be affected; see description.

Description: The CWAI will AND the contents of the CC register with the immediate byte and route the result back into the CC register. The AND operation allows one to selectively clear bits in the CC register. Then the E bit of the CC register is set and all the MPU registers are pushed onto the S stack. The MPU stops processing and waits for an interrupt. When an interrupt that is not masked out occurs, the MPU resumes processing at the vector address of that type of interrupt. This is a way of synchronizing the operation of the MPU with an external event.

For example, the S register has been loaded to establish the S stack; the instruction: CWAI #$EF will clear the I bit of the CC register, then set the E bit, and stack the entire set of MPU registers. The MPU stops and waits for an interrupt. In this case, the I bit has been cleared to allow an IRQ interrupt but the F bit may have previously been cleared, allowing an FIRQ interrupt. Also, the NMI interrupt may be processed since it can not be masked out. When an allowed interrupt occurs, IRQ for example, the MPU gets the vector address stored at addresses $FFF8 and $FFF9 and loads it into the PC register. The MPU starts processing instructions at the new address in the PC register. This new address is normally the starting address of an interrupt handler which, when finished, executes an RTI instruction. The RTI pulls all the MPU registers from the S stack and causes the MPU to resume processing at the instruction following the CWAI.

## DAA                                Decimal Addition Adjust

Source Forms: DAA

Operation: A' <- A+CF(MSN):CF(LSN)                Type: Arithmetic

CF is a correction factor for each BCD digit (nibble). CF, for each nibble, will be either a 0 or a 6 as determined below.

CF(LSN)=6 if 1)the H bit from the previous operation was set
or 2)the value of the LSN is greater than 9

CF(MSN)=6 if 1)the C bit from the previous operation was set
or 2)the value of the MSN is greater than 9
or 3)the MSN>8 and the LSN>9

Addressing Modes: Inherent.

Condition Codes:

H - Not affected.
N - Set if bit 7 of the resulting byte is set; cleared otherwise.
Z - Set if the resulting byte is zero; cleared otherwise.
V - Not defined (could be any state).
C - Set if a carry from bit 7 is generated or if the previous operation set the C bit; cleared otherwise.

Description: After adding BCD numbers with the ADDA or ADCA, the total may contain nonvalid BCD codes. The DAA is normally used immediately after an ADDA or ADCA because it uses the condition codes generated by them and operates only on the contents of the A register. The DAA performs the adjustment of the result of binary addition by adding a correction factor (0 or 6) to each nibble to make each nibble a valid BCD code. For example, the BCD numbers 55 and 17 can be added and adjusted with these two instructions (in this case the A register contains $55):

```
ADDA #$17
DAA
```

The ADDA will adds, in binary, $55 and $17 to yield $6C, which is routed back to the A register. The DAA inspects the contents of A and the CC register, that in this case will determine a CF(MSN) of 0 and a CF(LSN) of 6. The DAA adds the correction factors into the A register as:

```
  6 B        contents of A
+ 0 6        correction factors
-----
  7 2        valid BCD result in A
```

Now the A register contains $72, the correct result of adding the two BCD numbers. When adding BCD numbers composed of multiple bytes, the ADCA must be used to propagate any carries throughout the bytes being added.

## DEC

**Decrement**

Source Forms: DEC Q; DECA; DECB

Operation: M' <- M-1 or R' <- R-1 Type: Arithmetic

Addressing Modes: Inherent; Indexed; Extended; Direct.

Condition Codes:

- H - Not affected.
- N - Set if bit 7 of the resulting byte is set; cleared otherwise. N set indicates a negative result if using signed binary numbers.
- Z - Set if the resulting byte is zero; cleared otherwise.
- V - Set if an overflow or underflow occurred; cleared otherwise. This applies only to signed binary.
- C - Not affected.

Description: The DEC instruction will decrement the contents of a memory location (DEC Q) or the A or B (DECA or DECB) register by one. When decrementing straight binary values, only the BEQ or BNE should be used to branch based on the result, since the DEC instruction does not affect the C bit. If using signed binary numbers, any of the signed binary branches may be used. For example, the B register contains $C4. A DECB instruction will result in B containing $C3. In the CC register, N will be set and Z and V will be cleared.

## EOR

**Exclusive OR**

Source Forms: EORA P; EORB P

Operation: R' <- R ⊻ M Type: Logic

Addressing Modes: Immediate; Direct; Extended; Indexed.

Condition Codes:

- H - Not affected.
- N - Set if bit 7 of the resulting byte is set; cleared otherwise.
- Z - Set if the resulting byte is zero; cleared otherwise.
- V - Always cleared.
- C - Not affected.

Description: The EOR will read a byte from a memory address as specified by the operand field. That byte is exclusive ORed with the contents of the A or B (EORA or EORA) register and the result is routed back to the A or B register, respectively. For example, the B register contains $77. The EOR with immediate addressing: EORB #$0F results in the B register containing $78. The operation is demonstrated in binary as:

```
   0 1 1 1  0 1 1 1     contents of B
 ⊻ 0 0 0 0  1 1 1 1     immediate byte
 ------------------
   0 1 1 1  1 0 0 0     new contents of B
```

A bit set in the operand will toggle, or reverse, the state of the corresponding bit in the register.

## EXG — Exchange Registers

Source Forms: EXG R1,R2
Operation: R1 <-> R2 Type: Data Movement
Addressing Modes: Register.
Condition Codes: Not affected unless the CC register is exchanged.

Description: The contents of two registers, R1 and R2, are exchanged. The two registers must be of the same size: both 8-bit or both 16-bit. The registers that may be accessed are A, B, D, X, Y, U, S, CC, DP, and PC.
For example, the U register contains $1120. The instruction: EXG U,PC results in the U register containing what was the current PC register value and the PC register containing $1120. Now the MPU will start executing instructions address $1120. This example is a unique type of branch operation.

## INC — Increment

Source Forms: INC Q; INCA; INCB
Operation: M' <- M+1 or R' <- R+1 Type: Arithmetic
Addressing Modes: Inherent; Indexed; Extended; Direct.
Condition Codes:
H - Not affected.
N - Set if bit 7 of the resulting byte is set; cleared otherwise. N set indicates the result is negative if using signed binary numbers.
Z - Set if the resulting byte is zero; cleared otherwise.
V - Set if an overflow or underflow occurred; cleared otherwise. This applies only to signed binary.
C - Not affected.

Description: The INC will increment the contents of a memory location (INC Q) or the A or B (INCA or INCB) register by one. When incrementing straight binary numbers, only BEQ or BNE should be used to branch on the result since INC does not affect the C bit. If using signed binary numbers, any of the signed binary branches may be used. For example, the A register contains $D5. The INCA instruction will result in A containing $D6. The CC register will also be modified where the N bit is set and the Z and V bits are cleared.

## JMP — Jump to Effective Address

Source Forms: JMP EA
Operation: PC' <- EA Type: Miscellaneous
Addressing Modes: Direct; Extended; Indexed.
Condition Codes: None are affected.

Description: The effective address, as specified by the operand field, is loaded into the PC register causing the MPU to start executing instructions at the effective address. For example, the JMP with extended addressing: JMP $2122 will cause the MPU to jump, or unconditionally branch, to address $2122.

## JSR — Jump to Subroutine at Effective Address

Source Forms: JSR EA

Operation: S' <- S-1, (S) <- $PC_L$ Type: Miscellaneous

S' <- S-1, (S) <- $PC_U$

PC' <- EA

Addressing Modes: Direct; Extended; Indexed.

Condition Codes: None are affected.

Description: The JSR will first push the contents of the PC register onto the S stack. Then the PC register is loaded with the effective address, as specified by the operand field. The MPU starts executing instructions at the effective address, usually the starting address of a subroutine. The last instruction of the subroutine should be an RTS, which will pull the PC contents from the S stack, causing the MPU to resume processing at the instruction following the JSR. For example, the Y register contains $3E20. The JSR with indexed addressing: JSR 10,Y causes the MPU to stack the PC contents in the S stack and then start executing a subroutine at address $3E2A, the sum of the contents of Y and 10, decimal.

## LD (8 Bit) — Load Register from Memory

Source Forms: LDA P; LDB P

Operation: R' <- M Type: Data Movement

Addressing Modes: Immediate; Direct; Indexed; Extended.

Condition Codes:

H - Not affected.

N - Set if bit 7 of the byte read from memory is set; cleared otherwise. N set indicates a negative signed binary number has been loaded into the register.

Z - Set if a value of zero is loaded into the register; cleared otherwise.

V - Always Cleared.

C - Not affected.

Description: The LD instruction reads a byte from a memory address specified by the operand field. That byte is loaded, or routed, into the A or B (LDA or LDB) register. For example, LD with immediate addressing such as LDB #$8C results in the B register containing $8C and, in the CC register, N is set and Z and V are cleared.

## LD (16 Bit) — Load Register from Memory

Source Forms: LDD P; LDX P; LDY P; LDU P; LDS P
Operation: R' <- M:M+1 Type: Data Movement
Addressing Modes: Immediate; Direct; Extended; Indexed.
Condition Codes:

- H - Not affected.
- N - Set if bit 15 of the loaded register is set; cleared otherwise. N set indicates a negative 16-bit signed binary number has been loaded into the selected register.
- Z - Set if a value of zero is loaded into the register; cleared otherwise.
- V - Always cleared.
- C - Not affected.

Description: The LD instruction reads two bytes from two memory addresses; the operand field specifies the first address. The two bytes are concatenated so that the first byte is now the upper half of a 16-bit word and the second byte is the lower half. That 16-bit word is then loaded into the selected register. For example, the U register contains the value $2D04. At addresses $2D04 and $2D05 are stored two bytes such as:

```
2D04  5F 8A
```

The LD with indexed addressing: LDY ,U loads the Y register with the two bytes the U register points to. In this case, Y is loaded with $5F8A and, in the CC register, N, Z, and V are cleared.

## LEA — Load Effective Address

Source Forms: LEAX; LEAY; LEAS; LEAU
Operation: R' <- EA Type: Data Movement
Addressing Modes: Indexed.
Condition Codes:

- H - Not affected.
- N - Not affected.
- Z - For LEAX and LEAY, set if a value of zero is loaded into the X or Y register; cleared otherwise. For LEAS and LEAU, Z is not affected.
- V - Not affected.
- C - Not affected.

Description: The LEA instruction will calculate the effective address specified by the operand field, but only by using the indexed addressing mode. The effective address will be loaded into the selected index register. No data will be read from memory at the effective address. For example, the A register contains $27 and the X register contains $1A10. The LEA instruction: LEAU A,X loads the effective address ($1A37), the sum of the contents of the A and X registers, into the U register. The A and X registers are not modified. Some other examples are:

| | |
|---|---|
| LEAX ,U | transfer U contents to X |
| LEAY 5,Y | increment contents of Y by 5 |
| LEAU -9,U | decrement contents of U by 9 |
| LEAX D,S | S plus D transfered to X |

The auto increment/decrement indexed mode is not used. Due to the internal operation of the MPU, LEAY ,Y+ for example, will not result in Y being incremented.

## LSL — Logical Shift Left

Source Forms: LSL Q; LSLA; LSLB
Operation: C <- $\boxed{7 \quad \leftarrow \quad 0}$ <- 0 Type: Logic
Addressing Modes: Inherent; Extended; Indexed; Direct.
Condition Codes:
- H - Undefined, therefore a branch should not be used to test its condition.
- N - Set if bit 7 of the resulting byte is set; cleared otherwise.
- Z - Set if the resulting byte is zero; cleared otherwise.
- V - Cleared if the original bits 6 and 7 were both set or both clear; set otherwise.
- C - Set if bit 7 of the original byte was set; cleared otherwise.

Description: The LSL shifts the contents of a memory location (LSL Q), or the A or B (LSLA or LSLA) register, to the left one bit position. Bit 7 of the original byte is shifted into the C bit and bit 0 of the result is cleared. For example, address $310A contains $8A. The LSL with extended addressing: LSL $310A reads the byte, $8A, from address $310A. Then it is logically left shifted as seen below:

```
          1 0 0 0  1 0 1 0          byte from memory
    C <-  1 0 0 0  1 0 1 0  <- 0     shift left
   C=1    0 0 0 1  0 1 0 0           resultant byte
```

The resulting byte, $14, is stored in address $310A and in the CC register C and V are set and Z and N are cleared.

## LSR — Logical Shift Right

Source Forms: LSR Q; LSRA; LSRB
Operation: 0 -> $\boxed{7 \quad \rightarrow \quad 0}$ -> C Type: Logic
Addressing Modes: Inherent; Extended; Direct; Indexed.
Condition Codes:
- H - Not affected.

N - Always cleared.
Z - Set if the resulting byte is zero; cleared otherwise.
V - Not affected.
C - Set if bit 0 of the original byte was set; cleared otherwise.

Description: The LSR will shift the contents of a memory location (LSR Q), or the A or B (LSRA or LSRB) register, to the right one bit position. Bit 7 of the result is cleared and bit 0 of the original byte is shifted into the C bit. For example, the B register contains $9E. The LSR with inherent addressing: LSRB results in the following binary operation:

```
            1 0 0 1  1 1 1 0        old contents of B
      0 ->  1 0 0 1  1 1 1 0  -> C  right shift
     C=0    0 1 0 0  1 1 1 1        new contents of B
```

The B register now contains $4F and, in the CC register, C, N, and Z are cleared.

## MUL — Multiply

Source Forms: MUL
Operation: D' <- A x B                                   Type: Arithmetic
Addressing Modes: Inherent.
Condition Codes:
H - Not affected.
N - Not affected.
Z - Set if the result, in D, is zero; cleared otherwise.
V - Not affected.
C - Set if the operation sets bit 7 of the D register; cleared otherwise.

Description: The MUL will multiply (using straight binary) the contents of the A register by the contents of the B register. The resulting 16-bit word is routed into the D register. For example, A contains $4B and B contains $0C. The multiply instruction: MUL calculates the product of $4B and $0C, and stores that result in the D register. This can be demonstrated, in binary, as:

```
                 0 1 0 0 1 0 1 1         contents of A
               x 0 0 0 0 1 1 0 0         contents of B
               ------------------
               0 1 0 0 1 0 1 1
             0 1 0 0 1 0 1 1
  -------------------------------
  0 0 0 0 0 0 1 1 1 0 0 0 0 1 0 0         new contents of D
```

As a result, D contains $0284 and, in the CC register, Z is cleared and C is set. The D register is composed of the A and B registers linked together; therefore, the A and B contents are destroyed. The A register is the upper half and B is the lower half of the D register.

## NEG — Negate

Source Forms: NEG Q; NEGA; NEGB
Operation: M' <- 0-M or R' <- 0-R    Type: Arithmetic
Addressing Modes: Inherent; Indexed; Extended; Direct.
Condition Codes:
- H - Undefined, therefore a branch should not be used to test its condition.
- N - Set if bit 7 of the resulting byte is set; cleared otherwise.
- Z - Set if the resulting byte is zero; cleared otherwise.
- V - Set if the original byte was $80; cleared otherwise.
- C - Set if a borrow out of bit 7 of the ALU was generated during the subtraction process; cleared otherwise.

Description: A byte is read from a memory (NEG Q) location or the A or B (NEGA or NEGB) register. The two's complement of that byte is generated by subtracting it from zero and the resulting byte is routed back to its source. This instruction will reverse the sign of a signed binary number in a byte. There are two exceptions to this process, however: If the original byte contains $80, the result is $80 with the V bit set. If the original byte contains a zero, the result is zero and, only in this case is the C bit cleared.

For example, the B register contains $FC, the signed binary equivalent of decimal -4. The negate instruction: NEGB loads the B register with the two's complement of its original contents as seen below.

```
          b  b b b b  b
             0 0 0 0  0 0 0 0       zero value
          -  1 1 1 1  1 1 0 0       contents of B
          ----------------------
   C=1       0 0 0 0  0 1 0 0       new contents of B
```

The B register now contains $04, the signed binary representation of decimal +4. In the CC register, C is set and N,V, and Z are cleared.

## NOP — No Operation

Source Forms: NOP
Operation: None    Type: Miscellaneous
Addressing Modes: Inherent.    Condition Codes: None are affected.

Description: The NOP, or no-op, instruction does nothing. The MPU will fetch the NOP op code, decode it, and then fetch the following instruction.

## OR — Inclusive OR Memory into Register

Source Forms: ORA P; ORB P
Operation: R' <- R ∨ M Type: Logic
Adresing Modes: Immediate; Indexed; Extended; Direct.
Condition Codes:
- H - Not affected.
- N - Set if bit 7 of the resulting byte is set; cleared otherwise.
- Z - Set if the resuling byte is zero; cleared otherwise.
- V - Always cleared.
- C - Not affected.

Description: The OR instruction reads a byte from a memory location specified by the operand field. That byte is ORed with the contents of the A or B (ORA or ORB) register and the result is routed back to the respective A or B register. This instruction can be used to selectively set bits in the A or B registers. Any bit set in the operand byte will result in the corresponding bit being set in the register. For example, the A register contains $40 and the immediate operand is $11. The following OR instruction: ORA #$11 results in bits 0 and 4 of the A register being set as seen below:

```
   0 1 0 0  0 0 0 0      contents of A
 ∨ 0 0 0 1  0 0 0 1      operand byte
   -------------------
   0 1 0 1  0 0 0 1      new contents of A
```

Now A contains $51 and, in the CC register, the N, Z, and V bits are cleared.

## ORCC — Inclusive OR Memory into the CC Register

Source Forms: ORCC #xx
Operation: CC' <- CC ∨ MI Type: Logic
Addressing Modes: Immediate.
Condition Codes: Any bits may be set; see description.

Description: The ORCC instruction will inclusive OR the immediate byte with the contents of the CC register. The resulting byte is routed back to the CC register. This instruction lets one set selective bits in the CC register. Bits set in the operand byte will set corresponding bits in the CC register. For example, the CC register contains $22. The instruction: ORCC #$50 sets bits 6 and 4 in the CC register as seen below:

```
   0 0 1 0  0 0 1 0      contents of CC
 ∨ 0 1 0 1  0 0 0 0      operand byte
   -------------------
   0 1 1 1  0 0 1 0      new contents of CC
```

## PSHS Push Registers on the Hardware Stack

Source Forms: PSHS register list

Postbyte: $\boxed{7 \quad \text{push order} \rightarrow \quad 0}$ Type: Data Movement

Operation: If bit 7 of postbyte is set; S' <- S-1, (S) <- $PC_L$
S' <- S-1, (S) <- $PC_U$
If bit 6 of postbyte is set; S' <- S-1, (S) <- $U_L$
S' <- S-1, (S) <- $U_U$
If bit 5 of postbyte is set; S' <- S-1, (S) <- $Y_L$
S' <- S-1, (S) <- $Y_U$
If bit 4 of postbyte is set; S' <- S-1, (S) <- $X_L$
S' <- S-1, (S) <- $X_U$
If bit 3 of postbyte is set; S' <- S-1, (S) <- DP
If bit 2 of postbyte is set; S' <- S-1, (S) <- B
If bit 1 of postbyte is set; S' <- S-1, (S) <- A
If bit 0 of postbyte is set; S' <- S-1, (S) <- CC

Addressing Modes: Register.

Condition Codes: None are affected.

Description: The PSHS can push any, all, or none of the MPU registers onto the S (hardware) stack, but not the S register. The registers to be pushed are determined by the bits set in the postbyte, and the order of pushing is always the same (see Fig. 4-2). That order is: bit 7 first to bit 0 last, as seen in operation diagram. A byte is stored in the stack by decrementing the pointer, S, by one and then storing the byte at the address now contained in S. In a statement the registers to be pushed are contained in the operand field and are represented by these abbreviations: PC, U, Y, X, D, DP, B, A, and CC. Each abbreviation must be separated by a comma. The registers may be presented in any order in the operand field but the actual pushing order is always the same, as determined by the postbyte.

For example, the contents of the U, A, and B registers can be pushed onto the S stack, where S contains $380A, the current address of the top of the stack. The push instruction: PSHS A,B,U will store the contents of U at $3808 and $3809, B at $3807, and A at $3806. The S register now contains $3806, the new top of stack address.

## PSHU Push Registers on the User Stack

Source Forms: PSHU register list

Postbyte: $\boxed{7 \quad \text{push order} \rightarrow \quad 0}$ Type: Data Movement

Operation: If bit 7 of postbyte is set; U' <- U-1, (U) <- $PC_L$
U' <- U-1, (U) <- $PC_U$
If bit 6 of postbyte is set; U' <- U-1, (U) <- $S_L$
U' <- U-1, (U) <- $S_U$
If bit 5 of postbyte is set; U' <- U-1, (U) <- $Y_L$
U' <- U-1, (U) <- $Y_U$
If bit 4 of postbyte is set; U' <- U-1, (U) <- $X_L$

$U' \text{ <- } U-1, (U) \text{ <- } X_U$
If bit 3 of postbyte is set; $U' \text{ <- } U-1, (U) \text{ <- } DP$
If bit 2 of postbyte is set; $U' \text{ <- } U-1, (U) \text{ <- } B$
If bit 1 of postbyte is set; $U' \text{ <- } U-1, (U) \text{ <- } A$
If bit 0 of postbyte is set; $U' \text{ <- } U-1, (U) \text{ <- } CC$
Addressing Modes: Register.
Condition Codes: None are affected.

Description: The PSHU can push any, all, or none of the MPU registers onto the U (user) stack, but not the U register. The registers to be pushed are determined by the bits set in the postbyte, and the order of pushing is always the same (see Fig. 4-2). That order is: bit 7 first to bit 0 last, as in the operation diagram. A byte is stored in the stack by first decrementing the pointer (U) by one, and then storing the byte at the address now contained in U. In a statement, the registers to be pushed are contained in the operand field and are represnetted by these abbreviations: PC, S, Y, X, D, DP, B, A, and CC. Each abbreviation must be separated by a comma. The registers may be presented in any order in the operand field but the actual pushing order is always the same, as determined by the postbyte.

For example, the contents of the X, A, and CC registers can be pushed on the U stack, where U contains $2216, the current address of the top of the stack. The push instruction: PSHU A,X,CC will store the contents of X at $2214 and $2215, A at $2213, and CC at $2212. The U register now contains $2212, the new top of stack address.

## PULS — Pull Registers from the Hardware Stack

Source Forms: PULS register list
Postbyte: $\boxed{7 \text{ <- pull order } 0}$ Type: Data Movement
Operation: If bit 0 of postbyte is set; $CC' \text{ <- } (S), S' \text{ <- } S+1$
If bit 1 of postbyte is set; $A' \text{ <- } (S), S' \text{ <- } S+1$
If bit 2 of postbyte is set; $B' \text{ <- } (S), S' \text{ <- } S+1$
If bit 3 of postbyte is set; $DP' \text{ <- } (S), S' \text{ <- } S+1$
If bit 4 of postbyte is set; $X_U' \text{ <- } (S), S' \text{ <- } S+1$
$X_L' \text{ <- } (S), S' \text{ <- } S+1$
If bit 5 of postbyte is set; $Y_U' \text{ <- } (S), S' \text{ <- } S+1$
$Y_L' \text{ <- } (S), S' \text{ <- } S+1$
If bit 6 of postbyte is set; $U_U' \text{ <- } (S), S' \text{ <- } S+1$
$U_L' \text{ <- } (S), S' \text{ <- } S+1$
If bit 7 of postbyte is set; $PC_U' \text{ <- } (S), S' \text{ <- } S+1$
$PC_L' \text{ <- } (S), S' \text{ <- } S+1$
Addressing Modes: Register.
Condition Codes: The CC register may be pulled from the stack and therefore possibly change its contents. If the CC register is not pulled from the stack, there is no change to the CC register.

Description: The PULS can pull any, all, or none of the MPU registers from the S (hardware) stack, but not the S register. The registers to be pulled are determined by the bits set in the postbyte and the order of pulling is always the same (see Fig. 4-2). That order is: bit 0 first to bit 7 last, seen in the operation diagram. A byte is pulled from the stack by first reading the byte from the address contained in the S register and then incrementing the contents of S by one. In a statement, the registers to be pulled are contained in the operand field and are represented by these abbreviations: PC, U, Y, X, D, DP, B, A, and CC. The registers may be presented in any order in the operand field but the actual pulling order is always the same, as determined by the postbyte.

For example, the U, B, and DP registers can be loaded, or pulled from, the S stack where S contains \$3938, the current address of the top of the stack. The pull instruction: PULS U,B,DP fetches the bytes from addresses \$3938 - \$393B and puts them in the U, DP, and B registers, in that order. The S register will have been incremented to \$393C, the new top of stack address.

## PULU — Pull Registers from the User Stack

Source Forms: PULU register list

Postbyte: $\boxed{7 \quad \text{<-} \quad \text{pull order} \quad 0}$ Type: Data Movement

Operation: If bit 0 of postbyte is set; CC' <- (U), U' <- U+1
If bit 1 of postbyte is set; A' <- (U), U' <- U+1
If bit 2 of postbyte is set; B' <- (U), U' <- U+1
If bit 3 of postbyte is set; DP' <- (U), U' <- U+1
If bit 4 of postbyte is set; $X_U$' <- (U), U' <- U+1
$X_L$' <- (U), U' <- U+1
If bit 5 of postbyte is set; $Y_U$' <- (U), U' <- U+1
$Y_L$' <- (U), U' <- U+1
If bit 6 of postbyte is set; $S_U$' <- (U), U' <- U+1
$S_L$' <- (U), U' <- U+1
If bit 7 of postbyte is set; $PC_U$' <- (U), U' <- U+1
$PC_L$' <- (U), U' <- U+1

Addressing Modes: Register.

Condition Codes: The CC register may be pulled from the stack and possibly change its contents. If the CC register is not pulled from the stack, there is no change to the CC register.

Description: The PULU can pull any, all, or none of the MPU registers from the U (hardware) stack, but not the U register. The registers to be pulled are determined by the bits set in the postbyte and the order of pulling is always the same (see Fig. 4-2). That order is: bit 0 first to bit 7 last, as in the operation diagram. A byte is pulled from the stack by first reading a byte from the address contained in the U register and then incrementing the contents of U by one. In a statement, the registers to be pulled are contained in the operand field and are represented by these abbreviations: PC, S Y, X, D, DP, B, A, and CC. The registers may be presented in any order in the operand field, but the actual pulling order is always the same, as determined by the postbyte.

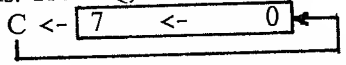
For example, the X, B, and DP registers can be loaded from the U stack where U contains $1770, the current address of the top of the stack. The pull instruction: PULU X,B,DP will fetch the bytes from addresses $1770 - $1773 and put them in the X, DP, and B registers, in that order. The U register will have been incremented to $1774, the new top of stack address.

## ROL — Rotate Left

Source Forms: ROL Q; ROLA; ROLB

Type: Logic

Operation:

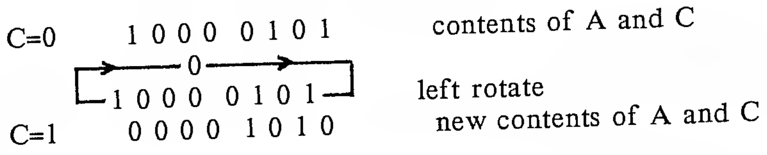


Addressing Modes: Inherent; Extended; Direct; Indexed.

Condition Codes:

- H - Not affected.
- N - Set if bit 7 of the resulting byte is set; cleared otherwise.
- Z - Set if the resulting byte is zero; cleared otherwise.
- V - Cleared if bits 6 and 7 of the original byte were either both set or both clear; set otherwise.
- C - Set if bit 7 of the original byte was set; cleared otherwise.

Description: A byte is read from a memory location (ROL Q) or the A or B (ROLA or ROLB) register and is shifted one bit position to the left. The original C bit is shifted into bit 0 of the result and bit 7 is shifted into the C bit. The resulting byte is routed back into its source. This is a left rotation of nine bits. For example, the A register contains $85 and the C bit is clear. The instruction: ROLA results in the A register conatining $0A and the C bit set. This operation can be seen, in binary, below:

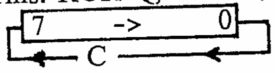


## ROR — Rotate Right

Source Forms: ROR Q; RORA; RORB

Type: Logic

Operation:



Addressing Modes: Inherent; Indexed; Direct; Extended.

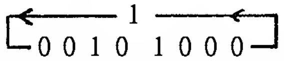
Condition Codes:

- H - Not affected.
- N - Set if bit 7 of the resulting byte is set; cleared otherwise.

Z - Set if the resulting byte is zero; cleared otherwise.
V - Not affected.
C - Set if bit 0 of the original byte was set; cleared otherwise.

Description: A byte is read from a memory location (ROR Q) or the A or B (RORA or RORB) register and is shifted one bit position to the right. The original C bit is shifted to bit 7 and the original bit 0 is shifted to the C bit. The resulting byte is then routed back to its source. This is a right rotation of nine bits. For example, the B register contains $28 and the C bit is set. The instruction: RORB will result in the B register containing $94 and the C bit clear. This can be seen as:

```
C=1      0 0 1 0  1 0 0 0      contents of B and C
```



```
                               right rotate
C=0      1 0 0 1  0 1 0 0      new contents of B and C
```

## RTI — Return from Interrupt

Source Forms: RTI

Operation: CC' <- (S), S' <- S+1, then, Type: Miscellaneous

If bit E is set; A' <- (S), S' <- S+1
B' <- (S), S' <- S+1
DP' <- (S), S' <- S+1
$X_U$' <- (S), S' <- S+1
$X_L$' <- (S), S' <- S+1
$Y_U$' <- (S), S' <- S+1
$Y_L$' <- (S), S' <- S+1
$U_U$' <- (S), S' <- S+1
$U_L$' <- (S), S' <- S+1
$PC_U$' <- (S), S' <- S+1
$PC_L$' <- (S), S' <- S+1

or if E is clear; $PC_U$' <- (S), S' <- S+1
$PC_L$' <- (S), S' <- S+1

Addressing Modes: Inherent.

Condition Codes: The CC register is loaded from the stack, so any or all bits may be changed.

Description: First the CC register is pulled from the S stack. If the E bit is set, the rest of the MPU registers will be pulled from the S stack, except for S. If the E bit is clear, the PC register will be pulled from the S stack. A byte is pulled from a stack by first reading the byte from the memory location contained in the pointer, S, and then S is incremented by one.

The RTI is usually the last instruction to be executed in an interrupt handler program. Normally an interrupt will cause the E bit to be set or cleared and then all (E=1) or just the PC and CC (E=0) registers are pushed on the stack. Then the MPU executes the interrupt handler which, upon completion, can return to the interrupted program by executing the RTI. The RTI causes the MPU to resume processing at the point of interruption. All the MPU's registers will contain their original contents, if E was set by the interrupt.

## RTS — Return from Subroutine

Source Forms: RTS
Operation: $PC_U$' <- (S), S' <- S+1 — Type: Miscellaneous
$PC_L$' <- (S), S' <- S+1
Addressing Modes: Inherent.
Condition Codes: Not affected.

Description: The RTS will pull the PC register from the S stack, causing the MPU to start executing instructions at the new address in the PC register. The RTS is typically the last instruction in a subroutine to be executed. The MPU is directed to a subroutine with a BSR or JSR, which stores the current PC contents in the S stack. The subroutine returns to the main program by executing the RTS, which loads the PC register with the return address from the S stack.

## SBC — Subtract with Borrow

Source Forms: SBCA; SBCB
Operation: R' <- R-M-C — Type: Arithmetic
Addressing Modes: Immediate; Direct; Extended; Indexed.
Condition Codes:

- H - Undefined, therefore it could be in any state.
- N - Set if bit 7 of the resulting byte is set; cleared otherwise. N set indicates a negative signed binary number.
- Z - Set if the resulting byte is zero; cleared otherwise.
- V - Set if an overflow or underflow occurred; cleared otherwise. This applies only to signed binary numbers.
- C - Set if a borrow is generated from bit 7 of the ALU; cleared otherwise.

Description: A byte is read from a memory location, specified by the operand field. That byte and the C bit, a borrow if set, are subtracted from the contents of the A or B (SBCA or SBCB) register. The resulting byte is routed back to its originating register. For example, address $1A07 contains $24, the A register contains $67 and the C bit is set. The instruction: SBCA $1A07 performs the following subtraction:

```
                     b      C is set
           0 1 1 0  0 1 1 1    contents of A
         - 0 0 1 0  0 1 0 0    operand byte
   C=0     0 1 0 0  0 0 1 0    new contents of A and C
```

The results: A contains $42 and C, N, Z, and V are cleared in the CC register.

## SEX — Sign Extend

Source Forms: SEX

Operation: Type: Arithmetic

If bit 7 of B is set, then A' <- FF
If bit 7 of B is clear, then A' <- 00

Addressing Modes: Inherent.

Condition Codes:

H - Not affected.
N - Set if bit 7 of the B register is set; cleared otherwise. N set indicates a negative signed binary number resulted.
Z - Set if the result is zero; cleared otherwise.
V - Not affected.
C - Not affected.

Description: All the bits of the A register are set equal to bit 7 of the B register. This instruction will cause the 8-bit signed binary number in the B register to be expanded into an equivalent 16-bit signed binary number in the D register. For example, the B register contains $E8, the signed binary equivalent of decimal -24. The instruction: SEX results in the D register containing $FFE8, the 16-bit signed binary equivalent of decimal -24. Remember that the D register is made up of the A and B registers; where the A register is the upper half of the D register. In the CC register, the N bit is set and the Z bit is cleared.

## ST (8 Bit) — Store Register into Memory

Source Forms: STA P; STB P

Operation: M' <- R Type: Data Movement

Addressing Modes: Direct; Extended; Indexed.

Condition Codes:

H - Not affected.
N - Set if bit 7 of the selected register is set; cleared otherwise.
Z - Set if the selected register contains zero; cleared otherwise.
V - Always cleared.
C - Not affected.

Description: The contents of the A or B (STA or STB) register are stored in the memory location specified in the operand field. For example, the B register contains $01. The instruction: STB $220A stores $01 at address $220A and, in the CC register, the N, Z, and V bits are cleared.

## ST (16 Bit) — Store Register into Memory

Source Forms: STD P; STX P; STY P; STU P; STS P
Operation: M':M+1' <- R  Type: Data Movement
Addressing Modes: Direct; Extended; Indexed.
Condition Codes:
- H - Not affected.
- N - Set if bit 15 of the selected register is set; cleared otherwise.
- Z - Set if the selected register contains zero; cleared otherwise.
- V - Always cleared.
- C - Not affected.

Description: The contents of a selected 16-bit register are stored in two consecutive memory locations; the operand field specifies the first location. For example, U contains $7D08. The instruction: STU $1A02 stores the upper half of U, $7D, at address $1A02 and the lower half, $08, at address $1A03. In the CC register, the N, Z, and V bits are cleared.

## SUB (8 Bit) — Subtract Memory from Register

Source Forms: SUBA P; SUBB P
Operation: R' <- R-M  Type: Arithmetic
Addressing Modes: Direct; Extended; Immediate; Indexed.
Condition Codes:
- H - Undefined, so it may be in any state.
- N - Set if bit 7 of the resulting byte is set; cleared otherwise. N set indicates a negative signed binary number.
- Z - Set if the resulting byte is zero; cleared otherwise.
- V - Set if an overflow or underflow occurred; cleared otherwise. This applies only to signed binary numbers.
- C - Set if a borrow is generated by bit 7 of the ALU; cleared otherwise.

Description: A byte is read from a memory location specified by the operand field. That byte is subtracted from the contents of the A or B (SUBA or SUBB) register and the result is routed to the selected register. For example, the A register contains $6A. The instruction with immediate addressing: SUBA #$27 results in A containsing $43. This subtraction process can be seen as:

```
                 b b b
            0 1 1 0  1 0 1 0        contents of A
          - 0 0 1 0  0 1 1 1        immediate byte
          ------------------
            0 1 0 0  0 0 1 1        new contents of A
```

In the CC register, the N, Z, and V bits are cleared.

## SUB (16 Bit) — Subtract Memory from Register

Source Forms: SUBD P

Operation: D' <- D-M:M+1 — Type: Arithmetic

Addressing Modes: Direct; Extended; Immediate; Indexed.

Condition Codes:

- H - Not affected.
- N - Set if bit 15 of the result in the D register is set; cleared otherwise. N set indicates a negative signed binary number.
- Z - Set if the result in D is zero; cleared otherwise.
- V - Set if a 16-bit overflow or underflow occurred; cleared otherwise. This applies only to signed binary numbers.
- C - Set if a borrow was generated by bit 15; cleared otherwise.

Description: Two bytes are read from two consecutive memory locations and concatenated to form a 16-bit word. That 16-bit word is subtracted from the contents of the D register and the result is routed back to the D register.

For example, the D register contains $072F, and bytes $04 and $16 are stored at addresses $1500 and $1501. The instruction with extended addressing: SUBD $1500 results in the D register containing $0319. This operation can be seen as:

```
                    b
    0000 0111 0010 1111     contents of D
  - 0000 0100 0001 0110     memory operand
  -----------------------
    0000 0011 0001 1001     new contents of D
```

In the CC register, the C, N, Z, and V bits are cleared.

## SWI — Software Interrupt

Source Forms: SWI

Operation: Set the E bit then; — Type: Miscellaneous

S' <- S-1, (S) <- $PC_L$
S' <- S-1, (S) <- $PC_U$
S' <- S-1, (S) <- $U_L$
S' <- S-1, (S) <- $U_U$
S' <- S-1, (S) <- $Y_L$
S' <- S-1, (S) <- $Y_U$
S' <- S-1, (S) <- $X_L$
S' <- S-1, (S) <- $X_U$
S' <- S-1, (S) <- DP
S' <- S-1, (S) <- B
S' <- S-1, (S) <- A
S' <- S-1, (S) <- CC
Set I and F bits

$PC' \leftarrow (\$FFFA):(\$FFFB)$

Addressing Modes: Inherent.

Condition Codes. The I and F bits of the CC register are set to mask out any IRQ and FIRQ interrupts. The E bit is also set.

Description: First the E bit is set to indicate that all the registers except S are stored in the stack. Then all the registers are pushed onto the S stack, as in the operation diagram. Each byte is pushed onto the stack by first decrementing the contents of S by one, then storing the byte at the new address in S. After stacking the registers, the I and F bits are set and the SWI vector address is read from addresses $FFFA and $FFFB. This vector address is in put into the PC register, causing the MPU to start executing an instruction at that address.

## SWI2 — Software Interrupt 2

Source Forms: SWI2

Operation: Set the E bit then; Type: Miscellaneous

$S' \leftarrow S-1, (S) \leftarrow PC_L$
$S' \leftarrow S-1, (S) \leftarrow PC_U$
$S' \leftarrow S-1, (S) \leftarrow U_L$
$S' \leftarrow S-1, (S) \leftarrow U_U$
$S' \leftarrow S-1, (S) \leftarrow Y_L$
$S' \leftarrow S-1, (S) \leftarrow Y_U$
$S' \leftarrow S-1, (S) \leftarrow X_L$
$S' \leftarrow S-1, (S) \leftarrow X_U$
$S' \leftarrow S-1, (S) \leftarrow DP$
$S' \leftarrow S-1, (S) \leftarrow B$
$S' \leftarrow S-1, (S) \leftarrow A$
$S' \leftarrow S-1, (S) \leftarrow CC$
$PC' \leftarrow (\$FFF4):(\$FFF5)$

Addressing Modes: Inherent.

Condition Codes: Only the E bit is set.

Description: First the E bit is set to indicate that all the registers except S are stored in the stack. Then all the registers are pushed onto the S stack, as in the operation diagram. Each byte is pushed onto the stack by first decrementing the contents of S by one, then storing the byte at the new address in S. After stacking the registers, the SWI vector address is read from addresses $FFF4 and $FFF5. This vector address is put into the PC register, causing the MPU to start executing an instruction at that address. Note that the I and F bits are not affected, the IRQ and FIRQ interrupts may or may not be masked out, depending on their previous state.

## SWI3 — Software Interrupt 3

Source Forms: SWI3

Operation: Set the E bit then; Type: Miscellaneous

```
            S' <- S-1, (S) <- $PC_L$
            S' <- S-1, (S) <- $PC_U$
            S' <- S-1, (S) <- $U_L$
            S' <- S-1, (S) <- $U_U$
            S' <- S-1, (S) <- $Y_L$
            S' <- S-1, (S) <- $Y_U$
            S' <- S-1, (S) <- $X_L$
            S' <- S-1, (S) <- $X_U$
            S' <- S-1, (S) <- DP
            S' <- S-1, (S) <- B
            S' <- S-1, (S) <- A
            S' <- S-1, (S) <- CC
            PC' <- ($FFF2):($FFF3)
```

Addressing Modes: Inherent.

Condition Codes: Only the E bit is set.

Description: First the E bit is set to indicate that all the registers except S are stored in the stack. Then all the registers are pushed onto the S stack as in the operation diagram. Each byte is pushed onto the stack by first decrementing the contents of S by one, then storing the byte at the new address in S. After stacking the registers, the SWI vector address is read from addresses $FFF2 and $FFF3. This vector address in put into the PC register, causing the MPU to start executing an instruction at that address. The I and F bits are not affected, so the IRQ and FIRQ interrupts may or may not be masked out, depending on their previous state.

## SYNC — Synchronize to External Event

Source Forms: SYNC

Operation: The MPU stops and waits until an interrupt occurs. Type: Miscellaneous

Addressing Modes: Inherent.

Condition Codes: May be affected by unmasked interrupt.

Description: This instruction causes the MPU to stop executing instructions and wait for an interrupt. Upon receiving an interrupt, if it is masked out or the interrupt signal is low for less than three cycles, the MPU resumes processing with the next instruction. A cycle refers to how often the pulses occur on the E and Q clock pins. In the Color Computer, a cycle is about 1.12 microseconds, or 1.12 millionths of a second. Upon receiving an unmasked interrupt that lasts more than three cycles, the MPU will perform the interrupt sequence, causing the MPU to execute the interrupt handler program. When the MPU is in the stopped state, the address and data bus signals from the MPU are electrically disconnected from the address and data buses so some other device can use the buses.

For example, if one wanted to increment a counter every time an IRQ interrupt occurred and the interrupts occurred in very rapid succession, the following program could be used.

```
       ORCC #$10
WAIT   SYNC
       INC COUNT
       BRA WAIT
```

The ORCC sets the I bit, masking out the IRQ interrupt. The SYNC instruction causes the MPU to wait until an interrupt occurs. Upon detecting the interrupt, processing continues by incrementing the COUNT and branching back to the SYNC to wait for another interupt.

## TFR — Transfer Register to Register

Source Forms: TFR $R_1$,$R_2$
Operation: $R_2$ <- $R_1$ Type: Data Movement
Addressing Modes: Register.
Condition Codes: Not affected unless $R_2$, the destination register, is the CC register. In that case, any bits can be set or cleared in the CC register.

Description: The contents of the source register, $R_1$, are transferred into the destination register, $R_2$. Both registers must be of the same length, both eight or 16 bits long. The registers to be used are designated by their abbreviation in the operand field. The abbreviations are: A, B, CC, DP, D, X, Y, U, S, and PC. For example, the A register contains $8C and the DP register is clear. The instruction: TFR A,DP results in the A and DP registers containing $8C.

## TST — Test

Source Forms: TST Q; TSTA; TSTB
Operation: TEMP <- M-0 or TEMP <- R-0 Type: Test
Addressing Modes: Inherent; Extended; Indexed; Direct.
Condition Codes:
- H - Not affected.
- N - Set if bit 7 of the byte being tested is set; cleared otherwise.
- Z - Set if the byte being tested is zero; cleared otherwise.
- V - Always cleared.
- C - Not affected.

Description: A byte is read from memory (TST Q) or the A or B (TSTA or TSTB) register and a value of zero is subtracted from it so the ALU can set or clear bits in the CC register. Neither the memory nor accumulator contents are changed. All this instruction really tells us is whether the byte has bit 7 set (N=1) or any bits set (Z=0). For example, The A register contains $80. The instruction: TSTA results in the N bit set and the Z bit cleared.

## FIRQ

**Fast Interrupt Request**

Operation: If F bit clear, then:

Type: Hardware Interrupt

```
S' <- S-1, (S) <- $PC_L$
S' <- S-1, (S) <- $PC_U$
Clear E bit
S' <- S-1, (S) <- CC
Set F and I bits
PC' <- ($FFF6):($FFF7)
```

Addressing Modes: Inherent.

Condition Codes: The E bit is cleared and the F and I bits are set.

Description: The occurrence of an FIRQ interrupt, if the F bit is clear, will cause the MPU to start the FIRQ interrupt sequence upon completion of the currently executing instruction. In the interrupt sequence, first the contents of the PC register are pushed on the S stack. Then the E bit is cleared and the CC register is pushed on the S stack. The F and I bits are set to prevent any further FIRQ and IRQ interrupts. Finally, the FIRQ vector address, read from addresses $FFF6 and $FFF7, is loaded into the PC register, causing the MPU to start executing the instruction at that address. Since only the PC and CC registers are pushed on the stack, this interrupt sequence is the fastest.

## IRQ

**Interrupt Request**

Opeation: If I bit clear, then:

Type: Hardware Interrupt

```
S' <- S-1, (S) <- $PC_L$
S' <- S-1, (S) <- $PC_U$
S' <- S-1, (S) <- $U_L$
S' <- S-1, (S) <- $U_U$
S' <- S-1, (S) <- $Y_L$
S' <- S-1, (S) <- $Y_U$
S' <- S-1, (S) <- $X_L$
S' <- S-1, (S) <- $X_U$
S' <- S-1, (S) <- DP
S' <- S-1, (S) <- B
S' <- S-1, (S) <- A
Set E bit
S' <- S-1, (S) <- CC
Set I bit
PC' <- ($FFF8):($FFF9)
```

Addressing Modes: Inherent.
Condition Codes: The E and I bits are set.

Description: The occurrence of an IRQ interrupt (if the I bit is clear) will cause the MPU to start the IRQ interrupt sequence upon completion of the currently executing instruction. In the interrupt sequence, first the contents of all the MPU registers, except the S and CC, are pushed ont the S stack. Then the E bit is set and the CC register is pushed on the S stack. The I bit is set to prevent any further IRQ interrupts. Finally, the IRQ vector address, read from addresses $FFF8 and $FFF9, is loaded into the PC register, causing the MPU to start executing the instruction at that address.

## NMI

**Non Maskable Interrupt**

Type: Hardware Interrupt

Operation: $S' \leftarrow S-1, (S) \leftarrow PC_L$
$S' \leftarrow S-1, (S) \leftarrow PC_U$
$S' \leftarrow S-1, (S) \leftarrow U_L$
$S' \leftarrow S-1, (S) \leftarrow U_U$
$S' \leftarrow S-1, (S) \leftarrow Y_L$
$S' \leftarrow S-1, (S) \leftarrow Y_U$
$S' \leftarrow S-1, (S) \leftarrow X_L$
$S' \leftarrow S-1, (S) \leftarrow X_U$
$S' \leftarrow S-1, (S) \leftarrow DP$
$S' \leftarrow S-1, (S) \leftarrow B$
$S' \leftarrow S-1, (S) \leftarrow A$
Set E bit
$S' \leftarrow S-1, (S) \leftarrow CC$
Set I and F bits
PC' <- ($FFFC):($FFFD)

Addressing Modes: Inherent.
Condition Codes: The E, F, and I bits are set.

Description: The occurrence of an NMI interrupt will cause the MPU to start the NMI interrupt sequence upon completion of the currently executing instruction. In the interrupt sequence, first the contents of all the MPU registers, except S and CC, are pushed on the S stack. Then the E bit is set and the CC register is pushed on the S stack. The I and F bits are set to prevent any further IRQ or FIRQ interrupts. Finally, the NMI vector address, read from addresses $FFFC and $FFFD, is loaded into the PC register, causing the MPU to start executing the instruction at that address.

The NMI interrupt can not be masked out by setting any of the CC register bits. The only time it is prevented is after a RESET operation before the S register has been loaded. However, if an NMI occurs during this time, its occurence is remembered. When the S register is finally loaded, the NMI interrupt sequence will begin.

## RESET

Hardware Reset

Operation: SET I and F bits
DP' <- 00
PC' <- ($FFFE):($FFFF)

Type: Hardware Interrupt

Addressing Modes: Inherent.
Condition Codes: Only the F and I bits are set.

Description: A RESET will cause the MPU to start the RESET sequence. In the sequence, first the F and I bits are set. The DP register is cleared and the PC register is loaded with the RESET vector address, read from addresses $FFFE and $FFFF. Then the MPU starts executing the instruction at the vector address. The RESET is normally the first operation performed when a microcomputer is turned on. It is also initiated by pressing the RESET button on the Color Computer.

# CHAPTER 6

# Assembly Programming with EDTASM+

Assembly programming requires a technical knowledge of your computer; you must also use and understand the operation of a text editor, an assembler, and debugging aids.

The text editor is a program that controls the computer while one enters the text; displays the text on the screen; and lets you edit or modify previously entered text. You should be familiar with the BASIC text editor by which you enter or modify BASIC statements. In assembly language, the text editor is used to enter assembly language statements, or the source code. The text editor can usually store the source code on cassette tape or disk, and read source code from tape or disk into memory for display and/or further editing. There is also the capability of printing the source code on a printer.

The assembler is the program that will scan or read through the source code and generate object code. The assembler can be directed to put the generated object code in memory so the microcomputer can execute it, to see if the program works. It can also be directed to store the object code on tape or disk, so it can be loaded into memory and executed at some other time.

Debugging aids help the programmer determine why a program does not work as desired. It is inevitable that a program of any size will initially contain errors or bugs. Debugging aids provide ways to check or monitor a program's operation. Debugging aids often are able to display or change the contents of memory, control program execution, and store or read the object code to or from tape or disk.

Throughout this book we will use Radio Shack's EDTASM+ ROM pack to demonstrate editing, assembling, and debugging. It is similar to most other packages that provide for editing, assembling, and debugging, but supports storing and reading data to and from cassette tape only. When the EDTASM+ ROM pack is plugged in, it and Extended Color BASIC are structured as seen in Fig. 6-1. Each block represents a functional mode of operation; the arrows show how to get from one mode to another. The command that must be entered to cause transition to another mode appears on each arrow. Note that the assembler, upon completion of the assembly, automatically returns to the editor.

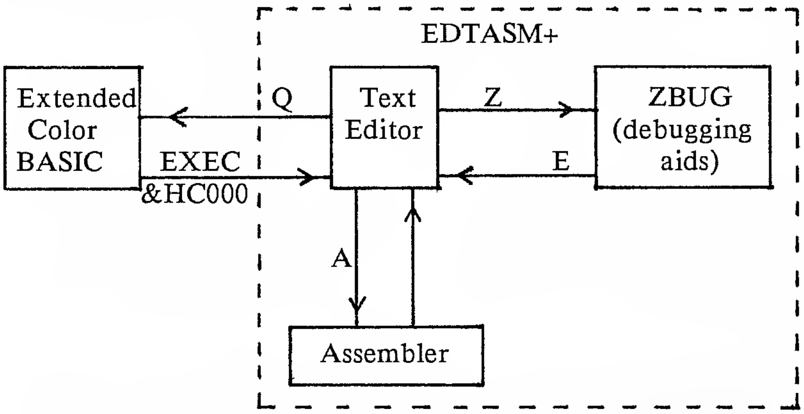


Fig. 6-1 Block Diagram of BASIC and EDTASM+ Functions.

Before writing any program one must have determined the sequence of steps or **algorithm** to be performed to arrive at a solution. This is especially important in assembly language programming because each instruction accomplishes very little. The technique of constructing a sequence of steps is called flowcharting. To familiarize yourself with or review flowcharts, refer to Appendix A. EDTASM+ will detect most errors when an improper entry is made, or some other problem arises. The EDTASM+ manual, Radio Shack catalog number 26-3250, provides a good explanation of the error messages in its Appendix D.

## THE TEXT EDITOR

The source code is entered from the editor mode. If the EDTASM+ module is plugged in, the Color Computer will be in the editor mode when first turned on. This can be verifed by observing the * prompt displayed at the lefthand edge of the screen. Source code is made up of a number of statements; each statement or line may contain up to five fields as follows:

```
Stmt No    Symbol    Cnd    Operand    Comment
```

The statement number is a whole number that can range from 0 - 63999. A series of statements are numbered so that the first statement has the lowest number and each suceeding statement has a higher number. Statement numbers do not have to be incremented by one for each consecutive statement. A series of statement numbers can be 10, 20, 30. When entering the statements, the editor will automatically supply the next statement number, thus relieving you of this task.

In the symbol field can be put the symbol, name, or label. The symbol may be up to six characters long; its first character must be alphabetic and the rest can be alphabetic or numeric. No special characters (such as %, #, or ?) may be in the symbol. The symbol will be assigned an address or value by the assembler, so the programmer does not have to use exact hexadecimal addresses. A symbol that has been defined (used once in the symbol field) can be used in the operand field to represent the desired address.

The command (or operation) field contains an instruction mnemonic or the mnemonic of a **pseudo-operation.** Pseudo-operations are commands to the assembler, covered later in this chapter. The instruction mnemonic in a command field directs the assembler to generate the machine code for that instruction.

The operand field may contain an address or the symbol of an address, or immediate data or the symbol for the immediate data to be used by the assembler to generate the the machine instruction operand field. The operand field may also contain abbreviations of registers used in indexed or register addressing.

In the comment field is the programmer's comment or reminder of what this particular instruction accomplishes.

When in the editor, start entering statements by using the **insert** (I) command. The editor will provide the first statement number, 100, and the first statement can be entered. The editor defaults to 100 or the next available number for the statement number, and an increment of 10 for succeeding statement numbers. The editor assumes that all numbers entered are decimal unless specified otherwise. Indicate a hexadecimal number by prefixing it wth $.

Now the symbol can be entered, if desired. Then the cursor can be positioned to the command field by hitting the right arrow or tab (->) key, where the mnemonic can be entered. After entering the mnemonic the statement can be terminated by hitting the ENTER key, or an operand can be entered with a space between it and the command. The statement can again be terminated by hitting the ENTER key, or the cursor can be moved to the comment field by hitting the right arrow key, where the comment can be entered. The statement is finally completed by hitting the ENTER key, which results in the presentation of the next statement number. Each field must be separated by a space or tab and the statement must be ended by the ENTER key. An example of this is:

```
I  <ENTER>     (invoke insert command)
00100  BEG    LDA #$20      LOAD CODE  <ENTER>
00110         CLRB          CLEAR COUNTER  <ENTER>
00120         STA $1203     STORE CODE  <ENTER>
00130         END   <ENTER>
00140  <BREAK>   (exit insert command)
```

These statements are now in the **buffer**, an area of memory reserved for holding entered text lines. The maximum allowable length of a statement line is 128 characters. Statement 130 contains the pseudo-command END, which tells the assembler that this is the end of the source code to be assembled. Upon completion of entering text, hitting the BREAK key will exit the insert command. When fields are to be skipped, as in statement 110, the right arrow key is pressed repeatedly until the cursor is at the desired field. Since only 32 characters can be displayed on one line with EDTASM+'s editor, statements will wrap around on the screen.

A statement which is all comment can also be entered by beginning it with an asterisk (*). After the asterisk any text can be entered. These comment lines usually serve to describe the program and are ignored by the assembler. For example, the following comment lines can be appended to the above statements:

```
I70  <ENTER>
00070  *THIS IS AN EXAMPLE OF  <ENTER>
00080  *USING THE TEXT EDITOR   <ENTER>
00090  **********************  <ENTER>
```

At this point a NO ROOM BETWEEN LINES message would be displayed, meaning that the next statement, 100, has already been entered. The 70 after the I tells the insert command that the next line to be entered is number 70. The insert command can also be used to enter text to be used for other purposes. For example, BASIC statements or just plain text can be entered; remember however that the assembler can not do anything meaningful with it.

The total listing can be displayed on the screen with the **print** (P) command:

```
P70:130   <ENTER>
```

This will list lines 70 - 130. Notice line 90; its purpose is to visually separate the descriptive text from the command statements.

The entered source code can be written to cassette tape using the **write** (W) command. This will record the text buffer contents using the Color Computer's ASCII code for each character. The W command can contain a name for the recording or file; the name must begin with an alphabetic character and may be up to eight characters long. If a name is not provided, it will be assigned the name NONAME. For example, make the cassette ready for recording and enter:

```
W EDITEST   <ENTER>
READY CASSETTE  <ENTER>
```

This will cause the statements in the buffer to be written to cassette tape with the name of EDITEST.

Source code should be saved on cassette tape before assembling and executing it since it can be easily lost. An assembly language program takes complete control of the microcomputer and the source code may be destroyed if there is a programming mistake in it.

The **verify** (V) command should be used to see that the text recording on cassette can be successfully read back. Position the cassette tape to the beginning point of the recording and ready it for playback. The verify command is initiated by typing:

```
V  <ENTER>
READY CASSETTE   <ENTER>
```

This causes the next file on tape to be checked for a valid recording. This is similar to BASIC's SKIPF command. If the recording is good, no message is printed. If the recording is faulty, a message will be printed on the screen. If that happens, try recording at a different place on the tape or on another tape.

The **delete** (D) command will delete selected statements from memory:

```
D70:130   <ENTER>
```

This causes our previously entered listing, statements 70 - 130, to be completely deleted.

The **load** (L) command is used to read a previously stored source code from cassette. First position the tape to the starting point of the recording and make it ready for playback. Then enter:

```
L  <ENTER>
READY CASSETTE  <ENTER>
```

The next file from tape will be read into the buffer area. The print command can be used to list what was just read in.

An interesting point is that files recorded with the editor's write command can be read using BASIC's CLOAD command. Also, files written with BASIC's CSAVE "XXX",A command can be read with the editor's load command.

**Text Editor Commands**

The following section will describe all EDTASM+ text editor commands, presented in alphabetical order. Special symbols will also be described. EDTASM+'s text editor is versatile in its wide array of commands and can be used for other purposes, such as entering and/or editing BASIC program statements, or as a line-oriented word processor.

■ Special Symbols

A colon is used to separate the beginning line number from the ending line number in a command that designates a range of lines to be acted upon. For example, the command to list lines on the screen: P100:200 will list text lines 100 - 200, if available.

The pound sign (#) represents the lowest line number that is in the buffer. For example: P#:150 will list text lines starting from the lowest numbered through line 150.

An asterisk represents the highest or last line number in the buffer. For example: P#:* will list all the text lines.

A period represents the current or most recently accessed text line number. For example: P100:. will list lines 100 through the current line.

An exclaimation point indicates that the following number states how many lines are to be acted upon. For example: P100!3 will list three lines starting with line number 100.

■ Going to BASIC

One can get to BASIC from the text editor by typing Q and the ENTER key. This initializes BASIC and displays its header on the screen. In the process, any text in the editor's buffer is deleted, but object code in areas of memory that BASIC does not use will not be deleted. From BASIC, control can be returned to the editor by entering:

```
EXEC &HC000      or      EXEC 49152
```

Either command causes the MC6809E to start executing instructions at address $C000, the starting address of EDTASM+. During transitioning from BASIC to EDTASM+, any BASIC statements in memory will be deleted.

■ Copy - Form: CXXX,YYY:ZZZ,TTT

The copy command will copy a range (YYY - ZZZ) of lines to a starting line number (XXX) with an increment of TTT. For example: C600,100:200,5 will copy lines 100 - 200 to the starting line number 600, and the line increment will be five. The original lines 100 - 200 are unchanged. An attempt to copy lines over others will not be executed, and an error message will be displayed.

■ Delete - Forms: DYYY:ZZZ or DYYY or DYYY!TT

The delete command will delete a statement or a series of statements from the buffer. For example: D210 will delete line number 210; D200:310 will delete lines 200 - 310; D100!5 will delete 5 lines, starting with 100.

■ Edit - Form: EYYY

The edit command will allow you to edit statement or line number YYY. For example: E120 causes line 120 to be displayed with the cursor positioned below it. Any editing subcommand can now be initiated to start editing that line. The editing subcommands are:

| | |
|---|---|
| A | Restarts the edit with the original unchanged line. |
| nCxxx.. | Changes n characters to string xxx.. starting at the cursored position. If n is omitted, only the cursored character is changed. |
| nD | Deletes n characters starting at the cursored position. If n is omitted, only the character at the cursored position is deleted. |
| E | Ends the editing process and enters any changes, but does not display the rest of the line. |
| ENTER | Ends the editing process and enters any changes, and displays the rest of the line. |
| H | Deletes rest of line from cursored position to the right and allows the addition of any new text. |
| Ixxx.. | Inserts the string xxx.. starting at the cursored position. Pressing the <- key will delete the character at a cursored position when in this subcommand. |
| nKx | Deletes all text from the cursored position to but not including the nth occurrence of character x. |
| L | Displays line being edited in its current form. |
| Q | Exits the edit mode and leaves the line in its original form. |
| X | Moves the cursor to the end of the line where more text can be entered. |
| Shift ↑ | Exits from edit subcommand. |
| n space bar | Moves cursor n positions to the right. If n is omitted, each depression of the spacebar will move the cursor one position to the right. |
| n <- | Moves the cursor n positions to the left. If n is omitted the cursor will move one position to the left for each depression of the <- key. |

■ Find - Form: Fxxx... or F

The find command will initiate a search for the text string xxx... starting with the line after the current line. It will stop each time the string is found and display the line containing the string. The search can be initiated again by entering F, which will use the last defined search string. If a search does not find the string, a message so indicating will be displayed. For example: FANDA will search for the string ANDA starting with the line after the current line.

■ Insert - Forms: I or IXXX or IXXX,TTT

The insert command opens the text buffer allowing text lines to be added or inserted. XXX specifies the starting line number, but if it is omitted, the editor will present 100 or the next available line number. TTT specifies the line increment, but if it is omitted, the editor will use 10 or the last specified increment.

■ Load - Forms: L or Lxxxxxxxx

The load command will read a text file named xxxxxxxx from cassette tape into the buffer. If a name is not provided it will read the next file on tape. The name can be up to eight characters long, and the first character must be alphabetic. An interesting feature of the load command is that it does not empty the buffer before loading it. Therefore, one can read a file that will be appended to the current contents of the buffer. This is useful for programs written in sections. A limitation is that the line numbers of the text read in must be different than the line numbers currently in the buffer.

■ Print - Forms: PXXX or PXXX:YYY or PXXX:TTT

The print command will list a line or series of lines in numerical order on the screen. A long listing can be temporarily halted by typing shift and @ concurrently, allowing certain lines to be read before they are scrolled off the screen. Listing is resumed by depressing any key except BREAK (which will cancel the print command). Examples of the print command are: P180 lists line 180; P160:330 lists lines 160 - 330; P205!62 lists 62 lines starting with line 205.

■ Printer Commands - Forms: TXXX or TXXX:YYY or TXXX!TTT and
HXXX or HXXX:YYY or HXXX!TTT

These commands will print the text in the buffer to a printer. The lines to be printed are specified the same as for the print (P) command.

The print (H) command will list specified lines in their entirety to the printer. The print (T) command will list specified lines without their line numbers to the printer. This command would be used if the text editor were being used as a word processor.

■ Renumber - Forms: N or NXXX or NXXX,TTT

The renumber command will assign new numbers and increments to all the lines in the buffer. XXX specifies the new starting line number and TTT specifies the new increment. If XXX is omitted, 100 or the current line number is used. If TTT is omitted, 10 or the current increment it used. For example: N300,20 will renumber all the lines so the first is 300 and the increment is 20.

■ Replace - Forms: R or RXXX or RXXX,TTT

The replace command allows one to replace line XXX and insert new lines with an increment of TTT. If the line number is omitted the replacement will start with the current line. If the increment is omitted, the replacement will use the last specified increment. For example: R120,2 allows one to reenter line 120 and insert new lines starting with number 122.

■ Verify - Forms: V or Vxxxxxxxx

The verify command will read the cassette text file named xxxxxxxx, checking to see that it was recorded without errors. If the file name is omitted it will verify the next text file on tape. If no errors are detected, no message is displayed. If errors are detected, a message so indicating is displayed.

■ Write - Forms: W or Wxxxxxxxx

The write command will record the text buffer contents on cassette tape with the file name xxxxxxxx. The name may be up to eight characters long and the first character must be alphabetic. If a file name is not provided, the editor will give it the name NONAME.

■ Going To ZBUG

Typing Z and the ENTER key will cause a transition from the editor mode to ZBUG, the debugging mode. The # prompt at the left hand edge of the screen indicates that ZBUG is in control. Typing E and ENTER, while in ZBUG, causes a transition to the editor mode. The contents of the text buffer will not be affected by either transition.

■ Scroll Up and Down

The scroll command will cause the previous or next text line to be displayed. Successive depressions of the up arrow key will cause the previous text line to be displayed followed by the lower numbered lines in declining numerical order. Successive depressions of the down arrow key will cause the next line to be displayed followed by the higher numbered lines in ascending numerical order.

## THE ASSEMBLER

The assembler can only be activated from the editor by the assemble (A) command. Upon activation, the assembler will read through the statements in the buffer, starting with the first or lowest numbered and stopping at the first encounter of a statement with an END command. The output of the assembler will be a listing of the source code, object code, and symbol table, and the object code in memory or on cassette tape.

**Assembling Into Memory**

Before assembling the source code must be in the text buffer. It can be put there by typing it in or by loading it from tape. In this case the previous example of source code will be used to demonstrate the assembler. The source code is:

```
00070 *THIS IS AN EXAMPLE OF
00080 *USING THE TEXT EDITOR
00090 **********************
00100 BEG     LDA #$20     LOAD CODE
00110         CLRB         CLEAR COUNTER
00120         STA $1203    STORE CODE
00130         END
```

Note that statement 130 contains the END command, telling the assembler to stop there.

The assembler is initiated in the editor with the assemble command and any of its options. The options let one control certain aspects of the assembly process. The options are presented in Table 6-1. To have the assembler put the generated object code in memory, the in-memory (IM) option is used:

```
A/IM   <ENTER>
```

The assembly process will be performed, a symbol table is built, a listing is displayed, and the object code put in memory.

The listing generated by the assembler provides much information about the assembled program. The listing resulting from assembling the above source code in memory is:

```
                00070 *THIS IS AN EXAMPLE OF
                00080 *USING THE TEXT EDITOR
                00090 **********************
08A6 86   20    00100 BEG     LDA #$20     LOAD CODE
08A8 5F         00110         CLRB         CLEAR COUNTER
08A9 B7 1203    00120         STA $1203    STORE CODE
     0000       00130         END
00000 TOTAL ERRORS

BEG    08A6
```

Unfortunately, the listing on the screen will not appear exactly like this because of wrap-around. The middle and right sides of the listing are just a repeat of the original source code, lines 70 - 130. To the left of each statement that contains a valid command mnemonic and operand is its object code. In the leftmost column is the exact hexadecimal address, or **absolute address**, where that machine instruction has been placed. At statement 100 is the absolute address of $08A6, the starting address where the LDA instruction is stored. In the next column is the op code of the LDA instruction; $86. The following column is the operand field of the instruction, $20, located at the absolute address $08A7. The next line down shows the single byte $5F, which is the CLRB instruction located at absolute address $08A8. The last instruction, STA, is located in addresses $08A9 - $08AB. The final END statement generates no object code, but does terminate the assembly process. This type of listing, with the source and object code on the same line, is called a **side-by-side** listing.

Immediately following the END statement appears the total number of errors the assembler detected in the source code. Almost always the errors are mistakes made entering the source code, such as misspellings, incorrectly used statement fields, a reference to undefined symbols, or defining a symbol more than once (multiply defined). A symbol is **defined** when it is used in the symbol field. A symbol is **referenced** when it is used in the operand field. The symbol BEG was defined in statement 110 but not referenced anywhere in the program. An assembly is not considered satisfactory if there are any errors, because the assembler will not assemble statements containing an error.

The last printed item is the symbol table. This is a list of all the symbols defined in the program and their assigned values. In this example there is only one symbol, BEG, the starting address of the program. In the symbol table BEG is followed by its asssigned value, $08A6. U will be printed to the right of an undefined symbol. M will be printed to the right of a multiply defined symbol. S will be printed to the right of a symbol defined by the SET pseudo-command (covered in the Pseudo-Commands section of this chapter). A symbol in the symbol field, if the mnemonic is that of a machine instruction, is always assigned the value of the address at which that machine instruction resides.

While the assembly process is taking place, the output listing scrolls up the screen quite quickly. You can halt it by quickly pressing the shift and @ keys together. The assembly will resume when you press any key except BREAK.

A useful option is the wait on error (WE) option. This will halt the assembly process when an error is detected so you can view the erroneous line on the screen. Hitting any key except BREAK will cause the assembly to resume to completion or until another error is detected. The command: A/IM/WE will begin assembling into memory and halt on detected errors. The assembler does not change the source code. One can initiate the assembler a number of times to see the screen listing. One can also edit source code mistakes and then assemble it again.

## Assembling To Cassette Tape

The assembler is directed to put the object code on tape by not using the in-memory (IM) option. A file name up to eight charcters long can be specified for the object code recording; the first character must be alphabetic. If a name is not specified, the assembler defaults to the name NONAME. For example:

```
A TEST1  <ENTER>
READY CASSETTE   <ENTER>
```

will cause the assembly process to be performed. The object code is recorded on cassette with the name TEST1, and the listing is put on the screen. The object code can later be loaded into memory using BASIC's CLOADM command or ZBUG's load (L) command. Using the previous assembly language example, the listing would appear as:

```
                     00070 *THIS IS AN EXAMPLE OF
                     00080 *USING THE TEXT EDITOR
                     00090 **********************
0000 86    20        00100 BEG     LDA #$20       LOAD CODE
0002 5F              00110         CLRB           CLEAR COUNTER
0003 B7    1203      00120         STA $1203      STORE CODE
           0000      00130         END
00000 TOTAL ERRORS

BEG       0000
```

Except for the address column at the extreme left, this listing is identical to that generated when assembling into memory. When assembling on tape the addresses of the instructions in the listing are **relative addresses**. An absolute address can not be assigned because the instructions are not in memory. The asssembler defaults to a starting address of 0000. The object code can be later loaded into memory starting at almost any address using the CLOADM command. The absolute addresses can then be manually calculated by adding the relative addresses to the load address. For example, the following command will load the object code into memory starting at address $2A10: CLOADM "TEST1",&H2A10. The absolute address of each instruction is found by adding its relative address to $2A10. The first instruction would be at $2A10, the second at $2A12, and the third at $2A13.

Other options can be used, such as the wait on error (WE): A TEST1/WE. This will stop the assembly at an erroneous statement. Again, an erroneous statement will not be assembled.

**The Assembler Options**

There are nine assembler options. We have discussed two of them, IM and WE. The LP, NL, NS, and SS options affect only the assembly listing, but the total errors will always be displayed or printed.

| Option | Description |
|---|---|
| LP | Put assembly listing on printer. |
| NL | Suppress listing except for symbol table. |
| NS | Delete symbol table from listing. |
| SS | Use short screen format for listing. |
| NO | Do not generate any object code. |
| IM | Assemble object code into memory. |
| WE | Wait on errors. |
| AO | Assemble object code at absolute address. |
| MO | Use manually entered addresses. |

Table 6-1 Assembler Options and Descriptions.

The LP option causes the listing to be printed on a printer connected to the serial port. Examples of using the LP option are:

```
A TEST/LP
A/IM/LP
```

The NL option suppresses the source code and object code sections of the listing so only the total errors and symbol table are displayed or printed. Examples are:

```
A/IM/WE/NL
A TEST2/NL
```

The NS option keeps the symbol table from being printed or displayed at the end of the listing. Examples are:

```
A/IM/NS
A TEST/NS
```

The short screen (SS) option causes the listing to be printed or displayed in a way that wrap-around is less likely to occur. The listing lines are formated so that the object code is put on one line and its associated source code is put on the next line. Examples are:

```
A TEST1/WE/SS
A/IM/WE/SS
```

The NO option causes the assembler to produce no object code. This option could be used to perform an initial check for source code errors. It can also be used to generate a spare listing on the printer. Examples are:

```
A TEST1/NO/WE
A/IM/NO
```

The IM option directs the assembler to put the generated object code in memory. That program can then be easily executed from ZBUG to see how it runs. First save the source code on tape so it will not be lost if the program destroys what is in the text buffer. If the IM option is not used the object code will be recorded on tape, from where it can later be loaded and executed.

The WE option directs the assembler to halt if a statement with an error is encountered. Assembling resumes after you hit any key except BREAK, which cancels the assembly process and returns to the editor.

The absolute origin (AO) option is used with the IM option only if an ORG command is in one of the statements. This option is not necessary when assembling to tape. The AO option causes the object code generated from the statements following the ORG statement to be put in memory starting at the address in the operand field of the ORG statement. An example is:

```
00100            ORG $2B1A
00110 START       CLRA
00120            STA $3001
00130            END
```

The ORG statement can not have a symbol. Now the following command can be entered: A/IM/AO. This causes the object code to be put in memory, starting at address $2B1A. (See the description of the ORG command in the Pseudo-Command section of this chapter for more information.)

When assembling to tape, this causes the object code generated from the statements following the ORG statement to be put on tape with an absolute address assigned to them. In this case, the side-by-side listing will contain absolute addresses.

The manual origin (MO) option, used only with the IM option, lets one specify where in memory the source statements and symbol table will be stored and where the object code will be put. This lets you put the object code where the source code and symbol table normally reside, or to use the graphics capabilities, which require the memory area in which the source code and symbol table normally reside. (This is discussed in more detail in the Technical Details section of this chapter.)

**Pseudo-Commands**

The pseudo-commands direct the assembler to perform operations beyond generating machine instructions. The pseudo-commands are represented by mnemonics used in the command field of a statement. They can perform the functions of controlling the assembler, defining symbols, putting data in memory, and reserving an area of memory for use by the assembled program.

The pseudo-commands do not use any of the MC6809E addressing modes. They have their own formats. The operand field may contain an expression using octal, decimal, or hexadecimal numbers, a string of text characters, or items involving calculations. The calculations that can be performed and their formats are the same as those in ZBUG. (For more information about the possible calculations, see the ZBUG section of this chapter.) A description of each pseudo-command and its mnemonic follows in alphabetical order.

■ END - Form: END EXPRESSION

The END command indicates that no more statements are to be assembled. This command is put at the end of a series of statements to be assembled. For example:

```
00100 START   LDX #$400
00110 ST1     STA ,X+
00120         END
00130         CMPX #$600
00140         BLO ST1
```

This causes only statements 100 - 120 to be assembled. When assembling on tape with an ORG command an expression can be put into the operand field to specify the address of the first instruction to be executed. This address will be stored in the object code file with the ORG address. The two addresses are used by BASIC's CLOADM and EXEC commands. For example:

```
00100         ORG $3010
00110 BEG     LDX #$400
00120         LDA #$20
00130 BE1     STA ,X+
00140         CMPX #$5FF
00150         BLS BE1
00160         RTS
00170         END $3015
```

The object code file generated with A NAME can be loaded and executed in BASIC with the following commands: CLOADM and EXEC. These will cause the program to be loaded at address $3010 and execution to start at the instruction in address $3015. No addresses have to be specified with the BASIC commands since they are read from tape.

■ EQU - Form: SYMBOL EQU EXPRESSION

The equate command defines a symbol by assigning to the symbol the value of the expression. All symbol values are made up of two bytes. The expression must be or result in a value that can be represented in two bytes. If the value is too large, the most significant bytes will be dropped until what is remaining will fit in two bytes. Sample statements are:

```
00100 AGE      EQU 88
00110 NUM1     EQU $7F2E
00120 PAGE     EQU 7*$0A
00130          LDA >AGE
00140          END
```

Assembling the example will result in a symbol table in which the symbol AGE equals $0058 (the equivalent of decimal 88), NUM1 equals $7F2E, and PAGE equals $0046 (the equivalent of the product of decimal seven and $A). Statement 130 is an example of referencing a symbol. The LDA with extended addressing will be assembled with an operand address which points to the two byte symbol AGE.

■ FCB - Form: SYMBOL FCB EXPRESSION

The FCB (fix contents of a byte) command stores the value of the expression in a memory location at the current address. The symbol is optional. If a symbol is provided, it is defined (put in the symbol table and assigned the value of the current address). A sample source code using the FCB command is:

```
00100 STR     CLRA
00110         STA  $2E01
00120 CNT     FCB $1F
00130         FCB 73
00140         END
```

When assembled, statement 120 will put $1F in memory immediately following the STA instruction, and the symbol CNT will be put in the symbol table with the value of that memory address. Statement 130 will result in $49 in the memory location after $1F. This is a poor program example since the MPU, upon completing the STA instruction, will try to execute the data $1F and $48.

■ FCC - Form: SYMBOL FCC DELIMITER STRING DELIMITER

The FCC (fix contents with characters) command stores characters between the delimiters, using ASCII, in memory starting at the current address. The delimiters identify the text string and can be any character not used in that text string. FCC can be used to store messages in memory to be displayed on the screen by the program. The symbol is optional. If used, it is defined.

```
00280          LDB >LINE
00290          JMP $201B
00300 LINE     FCC !ENTER TODAY'S DATE!
00310          END
```

When assembled, the ASCII codes of the characters in the text string will be put in memory immediately after the JMP instruction. Also, the symbol LINE will be put in the symbol table with the value of the starting address of the text string in memory.

■ FDB - Form: SYMBOL   FDB   EXPRESSION

The FDB (fix double byte) command stores the value of the expression in two consecutive bytes, starting at the current address. If a symbol is provided, it is defined. Examples are:

```
00210 YEAR     FDB 1983
00220 DAT      FDB YEAR.DIV.4
00230          END
```

Statement 210 results in $07BF, the equivalent of decimal 1983, stored in two memory locations immediately after the preceding instruction. Statement 220 results in the hexadecimal equivalent of the address the YEAR symbol equals divided by four with no remainder stored in the next two memory locations. Also the symbols YEAR and DAT will be assigned the addresses of their respective two-byte fields.

■ ORG - Form:          ORG   EXPRESSION

The ORG (originate) command indicates to the assembler where to originate the object code. The originating address is the value of the expression. When assembling into memory using the ORG command the AO option must be used. The object code of statements after an ORG statement is put in memory starting at the address specified by the expression. More than one ORG statement can be in the source code. An ORG command must be the first statement or the assembler will try to put the object code in memory starting at address zero, which is used by EDTASM+. For example:

```
00090          ORG $1000
00100          ORCC #$50
00110          LDX #$2000
00120          ORG $1700
00130          STA $3F02
00140          LDD ,X++
00150          END
```

This is assembled in memory with the A/IM/AO command; the object code of statements 100 and 110 is stored starting at address $1000 and the object code for statements 130 and 140 is stored starting at address $1700.

When assembling on cassette tape the ORG command will put the starting address, specified by the expression, at the beginning of the object code file. Then that object code file can be loaded with ZBUG's load (L) command or BASIC's CLOADM command without specifying the load address. The listing generated by the assembler will have absolute addresses in the address column. For example:

```
00100          ORG $1200
00110          LDA #$41
00120          LDX #$400
00130 ST1      STA ,X+
00140          CMPX #$440
00150          BLO ST1
00160          RTS
00170          END
```

This can be assembled on tape with the A NAME command; the AO option is not necessary. Then go to BASIC by hitting the Q and ENTER keys, and ready the cassette for reading the recorded file. The BASIC command: CLOADM will load the program into memory, starting at address $1200. It can be executed by using the command: EXEC &H1200.

■ RMB - Form: SYMBOL  RMB  EXPRESSION

The RMB (reserve memory bytes) command reserves a number of memory locations starting at the current address. The number of locations to reserve is specified by the expression. If a symbol is provided it will be defined. The RMB can be used to reserve an area of memory in the program where a table might be stored and/or manipulated. For example:

```
00300            LDX #TABL
00310            RTS
00320 TABL       RMB 22
00330            STA ,X
00340            END
```

In this example, 22 decimal locations between the RTS and STA machine instructions will be reserved. The symbol TABL will be the starting address of that area.

■ SET - Form: SYMBOL  SET  EXPRESSION

The SET command is similar to EQU except that a symbol can be set to a new value a number of times in one program. It also prints an S to the right of that symbol in the symbol table listing. This setting and resetting of the value only happens during the assembly process, not during the execution of the object code. For example:

```
00100 CNT    SET $1234
00110 TABL   EQU CNT.DIV.3
00120 CNT    SET $7200
00130        STA >CNT-$3000
00140        END
```

The symbol CNT is set to a value in statements 100 and 120. In statement 110, TABL is equated to $0611 ($1234 divided by decimal 3). In statement 130, the operand address is $4200 ($7200 minus $3000).

■ SETDP - Form: SETDP EXPRESSION

The SETDP command informs the assembler that any instructions using default extended addressing and whose most significant byte of the operand address equals the expression are to be assembled using direct addressing. However, it can not detect whether a symbol containing an address meets these specifications. Before using the SETDP command the DP register must be loaded with the appropriate value to get the expected results. Note also that the effect of the SETDP command can be overridden by preceding the operand with a greater than sign to force extended addressing. For example:

```
00100          LDB #$15
00110          TFR B,DP
00120          SETDP $15
00130          LDA $1566
00140          STX PAGE
00150          INC $2022
00160          STD >15FA
00170 PAGE     EQU $15F2
00180          END
```

Statements 100 and 110 prepare to use the direct addressing mode by loading the DP register with $15. Statement 130 will be assembled with direct addressing. The rest will not.

**Technical Details**

To get the most from EDTASM+ you must know how it works. EDTASM+ always uses the memory area extending from address 0000 - 05FF for its internal operations. Not every location is used, but be careful when modifying any. Within this area is the text **screen buffer**, extending from address $0400 - $05FF. In the screen buffer 512 decimal locations each correspond to a unique print position on the screen. Locations $0400 - $041F correspond to the 32 decimal print positions of the top-most line of the screen. A character code in location $0400 will cause that character to appear in the upper left corner of the screen. Locations $05E0 - $05FF correspond to the 32 decimal print positions of the bottom-most line of the screen. A character is displayed on the screen, if the video display generater (VDG) is in the text mode, by putting its **video display** code in a particular location of the screen buffer. The video display codes are another set of codes. Each code comprises one byte and corresponds to one printable character. Their only purpose is to display characters on the screen. (The complete set of video display codes used by the Color Computer can be found in Appendix C.)

EDTASM+ is in ROM occupying addresses $C000 - $DFFF. The starting address is $C000, where the MPU is directed to begin executing instructions when the Color Computer is turned on, if the EDTASM+ ROM pack is plugged in. The MPU can also be directed to this address from BASIC, ZBUG, or an assembled program.

| Address | Use |
|---|---|
| 0000 | Internal Operations |
| 0400 | Screen Buffer |
| 0600 | Internal Operations |
| 0800 | Text Buffer |
|  | Symbol Table |
|  | Object Code |
|  | Unused |
| 8000 | Extended Color BASIC ROM |
| C000 | EDTASM+ ROM |
| E000 | Unused |
| FF00 FFFF | Dedicated Addresses |

Fig. 6-2 Memory map with EDTASM+.

When source code is entered and assembled in memory, EDTASM+ must use more RAM to store this information. This working area normally starts at $0600. The area from $0600 - $0800 is used for internal operations. Normally the source code text is stored starting at address $0800 using ASCII codes to represent each character.

The EDTASM+ assembler is a two pass assembler: it scans or passes through the source code twice. On the first pass all the symbols in the source code are identified, assigned a value, and put in a symbol table. The symbol table is stored in memory immediately after the last source code statement. The symbol table comprises items composed of a string of ASCII characters corresponding to the symbol name followed by two bytes containing the symbol's value in straight binary. On the second pass the assembler generates the object code, directed by the mnemonic, operand, and any symbols in each statement. The object code is put in memory immediately after the symbol table, if assembling in memory and not using the AO option or ORG pseudo-command. Fig. 6-2 is a memory map depicting this memory usage. However, the AO option and ORG pseudo-command can be used to put the object code in any other part of memory except the areas used for the source code and symbol table and the area used by EDTASM+.

The area EDTASM+ normally uses, starting at $0600, can be changed: for instance, to make room for the high resolution graphic display mode which uses addresses $0600 - $1DFF. This is done by changing the contents of memory locations $00FF and $0100, also known as BEGTEMP, which specify the starting address of the working area. Normally BEGTEMP contains $0600, but other values can be put there. The two-byte value that can be entered is restricted such that the least significant byte must be 00.

This procedure must be done before the source code is entered or loaded in memory. The following operations are performed using ZBUG; you may want to read the ZBUG section of this chapter first. To set BEGTEMP to $1E00, enter the following commands.

```
B  <ENTER>                    select byte display mode
W  <ENTER>                    select word mode
FF/   600   1E00  <ENTER>     change BEGTEMP
```

EDTASM+ must now be reinitialized with the ZBUG command: GC006. The Go (G) command tells the MPU to start executing at the following address. Starting EDTASM+ at $C006 will not affect the contents of BEGTEMP. At this point the initial EDTASM+ display is on the screen and the source code can be entered and assembled in memory or on tape.

The above example will result in EDTASM+ using the area $1E00 - $2000 for internal operations and the source code text in memory will start at $2000 with the symbol table following it. If assembled into memory with the A/IM/WE command the object code will be put in memory immediately following the symbol table.

The assembler can also be set up to use USRORG. USRORG specifies the starting address in which to store the object code. The MO assembler option must be specified to use USRORG. USRORG is located at addresses 00FD and 00FE and normally contains the highest address available in RAM. Therefore, USRORG must be changed if the MO option is to be used.

The procedure is to use ZBUG to change USRORG and, if desired, BEGTEMP to the desired addresses. Then EDTASM+ is reinitialized, after which the source code can be entered and assembled using the MO and IM options. BEGTEMP must be set to a value lower than USRORG. One must be careful to allow enough room between BEGTEMP and USRORG for the working area, source code, and symbol table. A minimun of $0300 bytes is recommended, but more will be needed for larger programs. The number of bytes required for a particular program can be found by first assembling it in memory without any ORG statements and without the AO or MO options. Subtracting $0600 from the starting address of the object code will tell you the number of bytes required.

Suppose you wish to have the working area start at address $1E00 and the object code assembled in memory starting at $3000. First ZBUG is activated with the Z command, and the word display mode enabled with the W command. USRORG and BEGTEMP can be modified as seen below:

```
FD/    xxxx    3000  <ENTER>       change USRORG
FF/     600    1E00  <ENTER>       change BEGTEMP
```

Now EDTASM+ is reinitialized with GC006. At this time the EDTASM+ initial display is on the screen and the source code can be entered and assembled in memory using the MO and IM options: A/IM/MO/WE. This places the object code in memory starting at address $3000 and the working area starting at address $1E00. This has all been done without using the AO option.

## ZBUG DEBUGGING AIDS

ZBUG is the functional mode of EDTASM+ in which exist the debugging aids. One goes to ZBUG from the editor by hitting the Z and ENTER keys. The # prompt at the left edge of the screen indicates ZBUG is active. While in ZBUG all numbers typed are assumed to be hexadecimal unless specified otherwise. While in ZBUG, hitting E then ENTER will return you to the editor mode.

During a program's development and testing, one will spend a lot of time in ZBUG. Its facilities include many commands that assist in debugging a program. They can also be used to see how other programs work. The commands have been grouped into four general categories; inspect and change, program control, data movement, and calculations.

Inspect and change commands display and change the contents of specific memory locations. The program control commands provide for executing and controlling the execution of a program. Also included are commands that display and change the values in the MPU's registers. The data movement commands let one move data from place to place in memory and use a printer, tape, or the screen for viewing or saving areas of memory. The calculations section performs arithmetic and logical operations.

**Inspect And Change**

Memory contents can be displayed in one of four different display modes. After its content has been displayed one may change it to a different value.

■ Mnemonic Display - The default display mode is **mnemonic** (M); the contents of a series of memory locations are displayed as an instruction's mnemonic and operand field. ZBUG assumes that the display address typed is the starting address of an instruction and attempts to **disassemble** it and display the results. If the address does not point to an instruction the displayed results can be misleading. This display mode is activated by typing the M then ENTER keys, but this only need be done if another display mode had been previously specified. The contents of memory is displayed by typing the address followed by a slash. For example, let's display the contents of BASIC ROM:

```
8000/  ?? <  >
8001/  ASLB <  >
8002/  LDX #80DE <  >
8005/  LDU #12A <  >
8008/  LDB #0A
```

Notice that depressing the down arrow key (scroll down) causes the next instruction to be displayed. At address $8000 the contents do not match any instruction op code, so ?? is displayed. Actually, the contents of 8000 and 8001 are not instructions: ZBUG is just doing the best it can to interpret them as instructions. The display of addresses $8002 - $8008 are the actual instructions in ROM. After each instruction is displayed, the cursor is positioned to the right where one can enter a one-byte value that will be stored at the starting address of that instruction.

The up arrow (scroll up) key causes the displayed address to be decremented by one and displayed again. The same data is interpreted as different instructions because the pointer is specifying a different, and possibly nonvalid, instruction address.

If the displayed instruction is a branch or jump, hitting the right arrow (->) key will cause the display pointer to follow the branch or take on the value of the operand address and display its contents. This lets one follow the flow of a program.

■ Byte Display - The second display mode is the **byte mode** (B). It is activated by typing B then ENTER. Now the contents of a memory location will be displayed as a hexadecimal byte. Inspecting BASIC ROM again is done as follows:

```
B  <ENTER>          (select byte display mode)
8000/  45  <  >
8001/  58  <  >
8002/  8E  <  >
8003/  80  <  >
8004/  0DE
```

Here one sees the contents of the individual memory locations. At each line, when the cursor is to the right, one can enter a one-byte value to put in the displayed memory location. Depressing ENTER causes the entered value to be stored at the displayed location. Depressing the down arrow key instead of ENTER will store the new value in memory and display the contents of the next location.

■ Word Display - The third display mode is the **word mode** (W). It is activated by typing the W then ENTER keys. A pair of memory locations will be displayed on each line. For example:

```
W   <ENTER>          (select word mode)
8000/  4558  <  >
8002/  8E80  <  >
8004   0DECE
```

displays two bytes beginning with the specified display address. At each line, when the cursor is to the right, one can enter a two-byte value followed by the ENTER or  key that will put the value into the two displayed address.

■ ASCII Display - The fourth display mode is the **ASCII mode** (A), activated by typing the A then ENTER keys. This causes ZBUG to interpret the contents of a location as an ASCII code and display the character it represents. If the contents are not a valid ASCII code, nothing will be displayed. For example:

```
A   <ENTER>          (select ASCII mode)
8000/  E <  >
8001/  X <  >
8002/
```

displays the contents of the ROM addresses as if their contents were ASCII codes. As in the other display modes, one can change the contents of the displayed memory location. The contents of a location can also be changed to the ASCII code of a character by entering that character preceded with an apostrophe. This is applicable to all four display modes. For example:

```
        B   <ENTER>            (select byte mode)
        2000/   xx      'E  <ENTER>
```

will store $45, the ASCII code of E, in location $2000.

■ Numeric Base - One can direct displayed numbers to be displayed in hexadecimal, decimal, or octal. **Octal** numbers use base eight. The hexadecimal format is the default display format. This is done with the **output** (O) command followed by a 16, 10 or 8 to specify hexadecimal, decimal, or octal, respectively. For example: O10 <ENTER> results in all numbers displayed in decimal. The displayed number's base is indicated by a suffix; T indicates decimal, Q indicates octal, and no suffix indicates hexadecimal. Similarly, one can specify the base of all operator entered numbers with the **input** (I) command followed by 16, 10, or 8. The default base is 16. For example: I8 <ENTER> causes ZBUG to assume that any number typed is octal. The input command can be overridden by adding the suffix of H, T, or Q to an entered number to specify that this number is hexadecimal, decimal, or octal, respectively. For example: 150T/ displays the contents of decimal address 150 regardless of the input mode.

If a program has just been assembled in memory, ZBUG can display data using the values of symbols in the symbol table. For example, assemble the following source code into memory.

```
00100 STR     LDY #TABLE        GET TABLE POINTER
00110 ST1     LDA ,Y            GET CONTENTS
00120         BPL ST2           BRANCH IF NOT NEGATIVE
00130         NEGA              FORM TWO'S COMPLEMENT
00140         STA ,Y            PUT BACK IN TABLE
00150 ST2     LEAY 1,Y          INC TABLE POINTER
00160         CMPY #20+TABLE    END OF TABLE?
00170         BLO ST1           IF NOT, GET NEXT CONTENTS
00180         SWI               RETURN TO ZBUG
00190 TABLE   RMB 20            TABLE AREA
00200         END
```

After assembling the above, go to ZBUG. Now the object code can be displayed using a symbol instead of an absolute address. Display examples are:

```
        STR/   LDY #TABLE
        ST2/   LEAY 1,Y
```

■ Symbolic, Half Symbolic, and Numeric Submodes - When displaying object code with its symbol table in memory, there are three submodes of the mnemonic display mode. They are **symbolic** (S), **half symbolic** (H), and **numeric** (N); symbolic is the default display submode. The display example above is in the symbolic submode - the display addresses and operand addresses are displayed as their symbols. The half symbolic submode is selected by typing the H and ENTER keys. Subsequent display of the object code will put the display addresses in symbolic form and the operand addresses in numeric form. For example:

```
H    <ENTER>           (select half symbolic)
STR/   LDY #8B5  <   >
ST1/   LDA ,Y    <   >
ST1+2/   BPL 8AC
```

The numeric display submode is selected by typing the N and ENTER keys. Subsequent display of the object code will put all numbers and addresses in the numeric form. For example:

```
N    <ENTER>           (select numeric)
STR/   LDY #8B5  <   >
8A5/   STA ,Y    <   >
8A7/   BPL 8AC
```

Now the importance of labeling the first instruction of a program becomes obvious; the symbol can be used to find the start of the object code. The value of a symbol can be displayed by typing the symbol followed by an equal sign:

```
STR=  xxxx
```

where xxxx is its assigned numeric value (that will be displayed).

■ Alternate Displays - Three special display subcommands are activated with the semicolon, equals, and colon keys. The semicolon and equals keys are used when displaying object code in symbolic format. Pressing the semicolon key will cause that instruction to be displayed again in the numeric display submode. Pressing the equals key will cause the first byte of the displayed instruction to be displayed as a hexadecimal value. When displaying in any mode, pressing the colon key will cause ZBUG to interpret a byte's contents as if they were the contents of the CC register, and display one of the characters, EFHINZVC, for each corresponding bit that is set.

**Program Control**

The commands presented here are those one would use to test or execute a new program. During the following explanations the previous sample program will be used. This program will read the contents of each location in a table of 20 (decimal) bytes. If the value in a location is a negative signed binary number, it is negated or its two's complement is taken and that is stored in TABLE. Upon completion, the SWI instruction returns control to ZBUG. After this program is assembled into memory with no errors, one can go into ZBUG to run it.

■ Go Command - A program is executed from ZBUG with the Go (G) command. Its formats are:

```
Gxxxx <ENTER>  and  GSYMBOL <ENTER>
```

The xxxx is the starting address the MPU will be directed to. If the symbol table is in memory, the symbol of the instruction to be executed can be used. The sample program can be executed by entering GSTR. It will run very quickly and the SWI instruction will return control to ZBUG resulting in the display:

```
8 BREAK @ ST2+8
```

If desired, one can use ZBUG to inspect the locations in TABLE to see that they all contain positive numbers.

■ Single Step - One can make the MPU step through the program one instruction at a time with the **single instruction** (,) command. Its formats are:

```
xxxx,    and    SYMBOL,
```

The xxxx is an absolute address of an instruction. A symbol can be used if the symbol table is in memory. This will cause that one instruction to be displayed and executed. Subsequent instructions can be executed by repeatedly hitting the coma key. During this process one can view the sequence of instructions of a particular program. However, this feature does not work with instructions in ROM.

■ Register Display - At any time the execution of a test program is stopped and ZBUG is in control, the contents of all or any of the MC6809E internal registers can be displayed and/or altered. This is done with the **display register** (/) command prefixed with an R for all registers or a valid register abbreviation of one register. For example:

```
R/              (display all MPU registers)
DP/ xx          (display contents of DP register)
```

When a register's contents are displayed one can enter a new value to be put in, just like changing the contents of memory. However, when all the registers are displayed, none of their contents can be changed. The ability to view the contents of the registers is very useful when debugging a program.

■ Breakpoints - A **breakpoint** is a point in a program where one would like execution to stop so that interim results can be checked. However, a breakpoint can not be set for an instruction in ROM. A breakpoint is set or enabled with the X command. The X is entered with the address or symbol of the stopping point after it. Its formats are:

```
Xxxxx      and      XSYMBOL
```

A specific example is: XST1, which will set a breakpoint at the LDA ,Y instruction. Each time that instruction is encountered, the program will stop. One can then display memory or registers to see how the program is working. A maximun of eight breakpoints can be entered. They are numbered 0 - 7 in the order in which they were entered. An SWI instruction will cause a return to ZBUG at breakpoint number eight. When a breakpoint is encountered, ZBUG displays the breakpoint number and its address or symbol as:

```
0 BREAK @ ST1
```

After a breakpoint, the program can be single-stepped through the following instructions by hitting the coma key. Breakpoints that have been set can be listed with the **display breakpoint** (D) command. Typing the D then ENTER keys will display all the breakpoints.

Execution of a program stopped at a breakpoint can be resumed with the **continue** (C) command. Entering C will resume the program until a breakpoint is encountered again. Entering a Cxx such as: C5 will cause the program to continue executing up to the fifth (decimal) detection of a breakpoint, then stop.

**Data Movement**

Six ZBUG commands move data or object code within, to, or from memory. These commands work with blocks of memory. After typing a command, depress the ENTER key to perform it.

■ Move - Format: U XXXX YYYY NNNN

The move command copies the contents of a series of memory locations to another series of locations. The starting address of the area to be copied is specified by the XXXX field. The starting address of the destination area is specified by the YYYY field. NNNN specifies how many bytes to copy. The numbers entered in the fields are hexadecimal unless specified otherwise. If you are copying a program whose symbol table is in memory, a symbol can be used as an address. For example: U STR 3100 35 will copy $35 bytes of the sample program to an area of memory starting at address $3100. The program now resides in two places in memory.

■ Put - Format: P nnn... XXXX YYYY ZZZZ

This command saves the contents of an area of memory on cassette tape. What is saved may be object code or data. The file name, up to eight characters, is specified by the nnn... field. The XXXX field specifies the starting address and YYYY the ending address of the area to be put on tape. The ZZZZ field specifies the address of the first instruction to be executed after this tape file is loaded in memory. The XXXX, ZZZZ, and name fields are put in the file and used when loading it back in memory. If the symbol table is in memory a symbol can be used to specify an address. Examples of using the put command are:

```
P TESTA STR TABLE+20 STR
P TESTB 0123 423 2A2
```

An object code file can be loaded into address XXXX with BASIC's CLOADM or ZBUG's load command without specifying that address at load time. In BASIC the program can be run without specifying the start address. For example: CLOADM and EXEC will load and execute a properly recorded object code file made with the put command.

■ Verify - Format: Vnnn...

The verify command reads over an object code file generated with the put command or assembled onto tape to see that it was recorded without errors. When a name (up to eight characters) is provided in the nnn... field, it searches for that object code file, then verifies it. If the name is omitted, it will verify the next object code file on tape. This command can be used to ensure that a recording is good.

■ Load - Format: Lnnn...

The load command reads an object code file created with the put command or assembled onto tape. When a file name is provided, it searches for that file and loads it. If the name is omitted, the next file on tape is loaded. The object code or data will be loaded into memory at the starting address specified in the file. When loading a file created with the put command, the data will be loaded into memory at the starting address specified by the put command's XXXX field. When loading a file created by assembling on tape, the object code will be loaded in memory starting at address 0000 unless an ORG command was used. In that case the load address will be that specified by the ORG command.

■ Display - Format: T XXXX YYYY

The display command lists the contents of the range of memory addresses XXXX - YYYY on the screen. The contents are displayed in the format previously selected (word, byte, mnemonic, or ASCII). For example: T 800 850 displays the contents of addresses 800 - 850. The address can be hexadecimal, decimal, or octal. If a symbol table is in memory, a symbol can be used to specify an address.

■ Print - Format: TH XXXX YYYY

The print command lists the contents of the range of memory addresses XXXX - YYYY to a printer. The contents are displayed in the format previously selected (word, byte, mnemonic, or ASCII). For example: T 800 850 prints the contents of addresses 800 - 850. The address can be hexadecimal, decimal, or octal. If a symbol table is in memory, a symbol can be used to specify an address.

**Calculations**

ZBUG is capable of calculating the value of an expression composed of numbers, characters, parentheses, and operators. If there is a symbol table in memory, symbols may also be in an expression. The operators can be classified in three categories; arithmetic, logic, and relational. The arithmetic and logic operators and their actions are listed in Table 6-2. The relational operators are shown in Table 6-3.

| Arithmetic | | Logic | |
|---|---|---|---|
| Operator | Action | Operator | Action |
| + | Addition | .AND. | And |
| - | Subtraction | .OR. | Inclusive Or |
| * | Multiplication | .XOR. | Exclusive Or |
| .DIV. | Division | .NOT. | Complement |
| .MOD. | Modulus | < | Shift |

Table 6-2 Arithmetic and Logic Operators.

| Operator | Action |
|---|---|
| .EQU. | Test for Equality |
| .NEQ. | Test for Inequality |

Table 6-3 Relational Operators.

Values to be operated on are internally represented by two bytes of straight binary numbers. The results of any expression are also represented in two bytes. Thus, only integers are allowed. This allows values 0 - $FFFF which correspond to the maximum memory address range of the MC6809E. The two-byte size also exactly matches the number of bytes in which a symbol's value is represented. This allows expressions to be used in program statements where the calculated value will be assigned to a symbol or operand field.

From ZBUG the calculated value of an expression can be displayed by pressing the equal (=) key. For example:

```
2A+41= 6B
```

The base of numbers entered and displayed is controlled by the input and output commands. Base 16 is the default base for input and output. The base of inputted numbers can be set to ten as seen below.

```
I10 <ENTER>
98-35= 3F
```

Decimal 35 is subtracted from decimal 98 resulting in $3F. The input command can be overridden by adding a suffix to a number to specify its base. An H suffix indicates hexadecimal, T indicates decimal and Q indicates octal. For example:

```
98T-35H= 2D
```

demonstrates an expression with numbers of different bases.

If a calculation generates a number outside the range of 0 - $FFFF, the overflow bits are dropped and there is no indication of this condition. For example:

```
FF01+123= 24
```

If a result is negative, ZBUG attempts to represent it in two bytes. For example:

```
4-7= 0FFFD
```

However, any function that uses this value will interpret it as a positive number ($FFFD). This phenomenum is a type of wrap-around of the values represented in a limited number of bits.

The arithmetic operators are shown using hexadecimal input and output numbers. Addition, subtraction, and multiplication are straightforward. Just be sure the result will fit in two bytes of straight binary. Examples are:

```
A23+112= 0B35
E78-432= 0A46
3*104= 30C
1B*4A= 7CE
```

Leading zeros are not displayed. Numbers beginning with the hexadecimal digits of A - F are prefixed with a 0 to indicate it is a number and not a symbol or register.

Division is done normally, except, that remainders are discarded and not displayed:

```
        F.DIV.5= 3      ($F divided by $5)
     14A.DIV.2C= 7      ($14A divided by $2C)
```

The **modulus** is the remainder resulting from dividing the number before (.MOD.) by the number after (.MOD.):

```
F.MOD.5= 0
1E54.MOD.C5= 51
```

Logic operators are demonstrated using hexadecimal numbers. All entered numbers are represented in two bytes, even if they fit in one byte:

```
1234.AND.0F0F= 204
1234.OR.0F0F= 1F3F
1234.XOR.0F0F= 1D3B
.NOT.1234= EDCB
1234<2= 48C        (shift left two bit positions)
1234<-1= 91        (shift right one bit position)
```

The relational operators indicate whether an expression is true. If true, the resulting double byte is set to $FFFF. If false, the resulting double byte is set to 0000:

```
17.EQU.17= 0FFFF
23.EQU.123= 0
8A.NEQ.8A= 0
8A.NEQ.10A= 0FFFF
```

More than one operator can be in an expression. For example:

```
4*10A+7= 42F
1C23.DIV.0C.EQU.211= 0
```

When two or more operators are in an expression they are performed in a specific order. That order, from first to last is: *, .DIV., .MOD. <, .AND., .OR., .XOR., +, -, .EQU., .NEQ. The order in which the operations are performed can be changed by using parentheses. The operator enclosed in the largest number of parentheses will be performed first. For example:

```
((7E+1C).OR.2525)<3= 2DF8      (add first, shift last)
```

Characters can also be in expressions where their ASCII code is the numeric value used to calculate the resulting value. The ASCII code is specified by preceding a character with an apostrophe. For example: 'B = prints the ASCII code for B, $42. The following expression:

```
'B+25= 67
```

adds $25 to the ASCII code for B.

With ZBUG, one can use the Color Computer as a calculator to perform such tasks as converting numbers to different bases, or to perform arithmetic with hexadecimal, decimal, or octal numbers.

# CHAPTER 7

# Assembly Language Programming

This chapter describes assembly language programming techniques, conventions, and guidelines. Each assembly instruction is primitive, does little, and is unlike English, so guidelines have evolved to assist the programmer in writing a program, and to make reading source code listings easier and more informative.

Larger programs are usually written in sections; each section performs some specific task or group of tasks. After all sections are assembled and debugged, they are combined to form a complete program. The sections are categorized as the main, or core, program; subroutines; and interrupt handlers. Each section can also be made up of subsections. All sections should have comment lines describing what each does.

Many examples are provided to illustrate the important concepts and principles. You may find they are quite useful, and incorporate them into your library of programs.

## SOURCE CODE GUIDELINES

The source code specifies instructions and data, and also contains English comments describing the program. A well documented program can save a lot of time and frustration when debugging it or changing it later. Listing 7-1 is an example of a well documented program listing.

This program first clears the screen. In statement 180 the first address ($400) of the screen buffer is loaded into the X register. Then the video code for a space ($60) is loaded into the A register. The space code is put in each location of the screen buffer in program loop statements 200 - 220. Now preparation can be made to put the text line on the screen. The starting address ($4C0) of the eighth display line is loaded into the X register at line 230 and the starting address of the text (#CLTEXT) is loaded into the Y register at line 240. The loop in lines 250 - 320 performs a number of functions. The ASCII code for each text character is read from the data area at line 250. If the end of text code ($04) is detected at lines 260 and 270, the branch to line 330 is taken, ending the program. If not, it is determined at lines 280 and 290 whether the ASCII code is also a video display code for an uppercase character. This needs to be done because the ASCII code of a character is not necessarily its video code. This can be seen in Appendix D. If the code is greater than $40, it is a valid uppercase video text code and is put in the screen buffer at line 310. If it is not valid, it is converted to one at line 300, by adding $40 to it and then placing it in the screen buffer at line 310. Line 320 branches to the beginning of this loop to act on the next character.

```
00100 **********************************************
00110 *PROGRAM NAME: CLRPRN
00120 *BY SAM FASTCODE AUG 2,1985
00130 *THIS PROGRAM WILL CLEAR THE SCREEN AND DISPLAY
00140 *AN UPPER CASE TEXT LINE. IT IS TO BE EXECUTED FROM
00150 *BASIC TO WHICH IT WILL RETURN UPON COMPLETION.
00160 **********************************************
00170         ORG $2A00       LOADING ADDRESS
00180 CLRPRN  LDX #$400       GET SCREEN POINTER
00190         LDA #$60        GET SPACE CODE
00200 CL1     STA ,X+         PUT CODE ON SCREEN
00210         CMPX #$5FF      END OF SCREEN?
00220         BLS CL1         IF NOT, DO AGAIN
00230         LDX #$4C0       GET DISPLAY POINTER
00240         LDY #CLTEXT     GET TEXT POINTER
00250 CL2     LDA ,Y+         GET TEXT CODE
00260         CMPA #$04       END OF TEXT?
00270         BEQ CL4         IF SO, GO TO END
00280         CMPA #$40       GOOD VIDEO CODE?
00290         BHS CL3         IF SO, USE IT
00300         ADDA #$40       CHANGE TO VID CODE
00310 CL3     STA ,X+         PUT ON SCREEN
00320         BRA CL2         DO AGAIN
00330 CL4     RTS             RETURN TO BASIC
00340 **********************************************
00350 CLTEXT  FCC /TESTING LINE 0123456789 !"#$/
00360         FCB $04         END OF TEXT CODE
00370 **********************************************
00380         END $2A00       START EXECUTION ADDR
```

Listing 7-1 The CLRPRN program.

The comments in this program help immensely in understanding the program and don't cost anything in memory use or MPU time after the source code is assembled. The object code is exactly the same whether or not there are any comments; the assembler completely ignores comments.

There is more to documenting a source code than just commenting each statement. Lines 110 - 150 are the descriptive header, bracketed by lines 100 and 160 to set it off from the instruction statements. Line 110 gives the program name, CLRPRN (clear and print). The name is limited to six characters because it will be used as the name or label of the first instruction in the program. The name also specifies the starting address.

The source code and object code will be saved on tape. The file name should be the same as the program name with an indicater to tell whether that file is the source code or object code. Since the program name is limited to six characters and a file name may contain up to eight characters, two characters are available to use as an indicator. One technique adds an O or S suffix to the program name; the O indicates an object code file and the S a source code file. For example, the source code could be saved with the following command: W CLRPRNS. The object code would then be put on tape with the command: A CLRPRNO. There is still a character position open. It could be used to indicate the version of that program, since a program often goes through many revisions before it finally works as desired. Ten versions can be noted, zero through nine. The source and object code files would be saved with the following EDTASM+ commands: W CLRPRNS3 and A CLRPRNO3. It is best to use one tape for source code files and another for object code files.

Line 120 gives the author and completion date of the program which should be included if the program is of any value. It protects of your work and it is necessary if the program is to be copyrighted. Lines 130 - 150 describe the operation of the program, very handy at a later date when you have forgotten what the program does.

Lines 170 and 380 specify the load address and the address of the initial instruction of the program. They do not have to be the same value. Since this program is to be used from BASIC, line 170 tells the CLOADM command where to load it and line 380 tells the EXEC command where to start executing. This makes using the program much easier. After the object code is assembled onto tape, it can be easily loaded and executed with the following BASIC commands:

```
CLOADM "CLRPRNO"
EXEC
```

Lines 170 through 380 are all commented. You may not feel this is needed with shorter programs, but they may be combined together to yield a large program, and then you will be grateful for the comments.

The data area, lines 350 and 360, is separated from the rest of the program by lines of asterisks to make it easy to find. This also helps you avoid putting a data area within a sequence of instructions, causing the MPU to execute the data as if they were instructions.

The symbols used in this program are derived by a technique to keep one from defining a symbol twice. The multiply-defined symbol problem is most likely to arise in large programs composed of many sections. One may mistakenly define a symbol in one section and then again in another section. When the sections are assembled the assembler detects the error, which must then be corrected. The technique used here requires that each small program or program section be given a unique name, and the first two characters of each name must be different from the first two characters of any other name. All the symbols defined within a section are composed of the same two first letters plus up to four more characters. Thus, the sample program CLRPRN would define the symbols CLRPRN, CL1, CL2, CL3, CL4, and CLTEXT. The only labels that can be the same in a group of subprograms are those in the operand fields; the different sections may use or reference any data area created by any other section. Remember, the potential problem is defining a symbol twice, not using it twice.

**Segmented Programs**

A program can be built in segments that are assembled and tested separately before put together. Smaller sections are much easier to debug than a large program.

A sample program (PHEX) follows; its tasks are to clear the screen and display the hexadecimal digits contained in a byte in memory. The program is to be executed from ZBUG and on completion will return to ZBUG. The program is divided into sections with clearly defined tasks. The first section's task is to clear the screen. Its source code would look like Listing 7-2.

```
100 *CLEAR THE SCREEN
110 ***********************************************
120 PHEX     LDX #$400        START OF DISPL BUFFER
130          LDA #$60         SPACE CODE
140 PHCL     STA ,X+          PUT IN DISPL BUFFER
150          CMPX #$5FF       END OF BUFFER?
160          BLS PHCL         IF NOT,DO AGAIN
170          SWI              RETURN TO ZBUG
180 ***********************************************
190          END
```

Listing 7-2 Clearing the screen.

The first instruction is labeled PHEX, the name (print hex) of the total program. The section should be assembled and tested to be sure it works properly, then the source code stored on tape for later use.

The next section inspects each hexadecimal digit (nibble) of a byte and generates its video display code. The video codes are stored in memory for the next section to use. First, the most significant nibble of a byte is inspected, its video code generated, and put in memory. Then the least significant nibble is similarly dealt with. A flowchart depicting this procedure can be seen in Fig. 7-1. The only valid video codes are those for hexadecimal digits 0 - 9 and A - F, found in Appendix D. This task would be coded, tested and debugged, and its source code saved on cassette tape.

The source code for each section should use different line numbers so they won't overlap those of another section. For example, the first section uses lines 100 - 190. The second section should start with line 200.

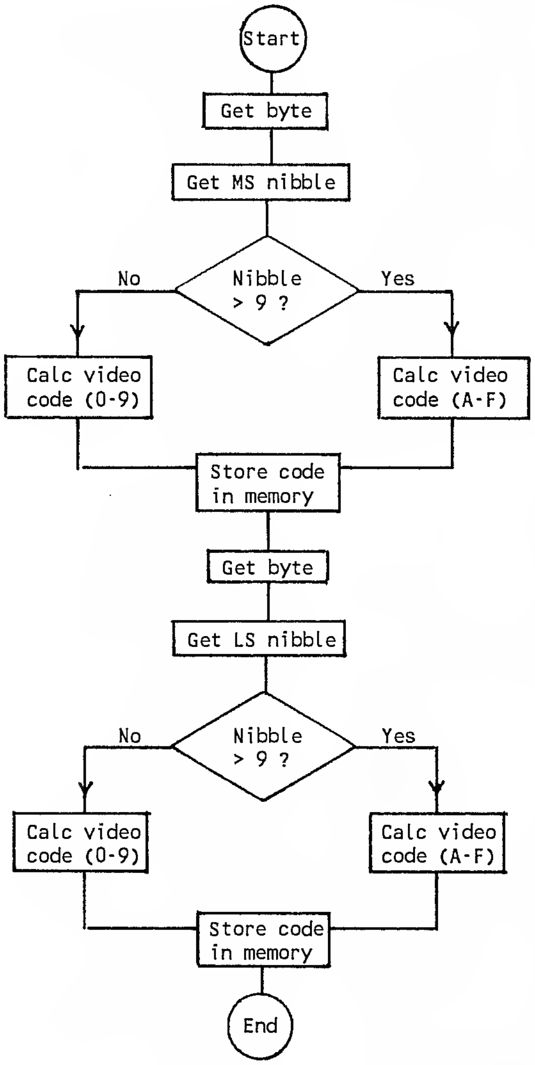


Fig. 7-1 Generating Video Codes of Hexadecimal Digits.

The source code for each section should use different line numbers so they won't overlap those of another section. For example, the first section uses lines 100 - 190. The second section should start with line 200.

The third section moves the video display codes, generated by the previous section, into the display buffer. In this case they are displayed on the fifth line, toward the right side of the screen.

```
100 ********************************************
110 *PROGRAM NAME: PHEX
120 *BY AUTHOR - DATE
130 *THIS PROGRAM IS TO BE EXECUTED FROM ZBUG TO
140 *WHICH IT WILL RETURN UPON COMPLETION. IT WILL
150 *CLEAR THE SCREEN, GENERATE THE VIDEO CODES OF
160 *THE HEX DIGITS IN A BYTE, AND DISPLAY THEM.
170 ********************************************
180         ORG $2000       LOAD ADDR
190 ********************************************
200 *CLEAR THE SCREEN
210 ********************************************
220 PHEX    LDX #$400       START OF DISPL BUFFER
230         LDA #$60        SPACE CODE
240 PHCL    STA ,X+         PUT IN DISPL BUFFER
250         CMPX #$5FF      END OF BUFFER?
260         BLS PHCL        IF NOT,DO AGAIN
270 ********************************************
280 *GENERATE THE VIDEO CODES OF THE HEX DIGITS
290 *IN A BYTE AT PHBYTE AND STORE THEM AT PHBYDS
300 ********************************************
310         LDX #PHBYDS     STORAGE ADDR
320         LDB PHBYTE      GET BYTE
330         LDA #$10        MULTIPLIER VALUE
340         MUL             SHIFT MS NIBBLE INTO A
350         CMPA #$09       GREATER THAN 9?
360         BHI PHCVA       IF SO,CONV TO ALPHA
370         ORA #$70        CONV TO NUM CODE
380         BRA PHCVB       GO STORE IT
390 PHCVA   ADDA #$37       CONV TO ALPHA CODE
400 PHCVB   STA ,X+         STORE CODE
410         LDA PHBYTE      GET BYTE
420         ANDA #$0F       GET LS NIBBLE
430         CMPA #$09       GREATER THAN 9?
440         BHI PHCVC       IF SO,CONV ALPHA
450         ORA #$70        CONV TO NUM CODE
460         BRA PHCVD       GO STORE IT
470 PHCVC   ADDA #$37       CONV TO ALPHA CODE
480 PHCVD   STA ,X+         STORE CODE
490 ********************************************
500 *PUT TWO VIDEO CODES AT PHBYDS INTO DISP BUFF
510 ********************************************
520         LDD PHBYDS      GET VIDEO CODES
530         STD $498        PUT INTO BUFFER
540         SWI             RETURN TO ZBUG
550 ********************************************
560 PHBYTE  FCB $7C         TEST BYTE
570 PHBYDS  RMB 2           STORAGE AREA
580         END $2000       EXEC ADDR
```

Listing 7-3 The PHEX Program.

After each section is written and debugged, their source codes can be merged together to form the total program. The three source code files can be successively loaded by the editor's load (L) command. If the source code statement numbers of each file do not overlap, the files will be appended to each other in the editor's text buffer. For example, the first section used lines 100 - 190, the second 200 - 500, and the third 600 - 700. Now the SWI and END statements must be removed from all but the last section. Any source code data areas of any section are then moved to follow the last section. A descriptive header can be added and the renumber (N) command used to organize the line numbers. The final result would look like Listing 7-3.

The total program should then be assembled and tested. The final working source code and object code can be saved on tape. The program is loaded from ZBUG with the load (L) command. It is executed by the G2000 command. The contents of PHBYTE, $7C, will be displayed on the right side of the screen. You can use ZBUG to put a different value in PHBYTE and then run the program again to see that it displays the new value. However, be careful to modify just that location so other data or instructions are not changed.

## SUBROUTINES

Many times tasks must be done a number of times. To conserve memory, those program sections should exist only once. The main program will execute it whenever that task needs to be done, and upon completion of that section the main program will resume execution. This type of a program section is a **subroutine**. Its relationship with the main, or calling, program can be seen in Fig. 7-2.

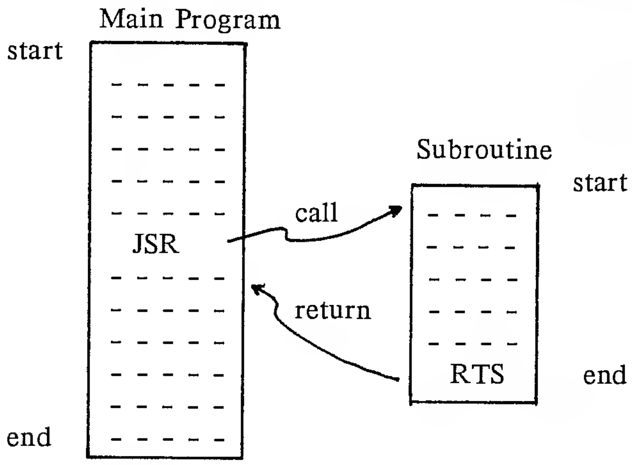


Fig. 7-2 Concept: Main Program and Subroutine.

The MPU is directed to a subroutine by a JSR or BSR instruction. This is known as **calling** a subroutine. The JSR or BSR cause the contents of the PC register to be pushed onto the S stack (saved), and execution starts at the operand address, the starting address of the subroutine. A complete description of these instructions is in Chapter 5. When the subroutine is complete, it executes its last instruction: RTS. RTS pulls the contents of the PC register from the S stack, causing the MPU to resume execution at the instruction immediately following the JSR or BSR in the main program.

Subroutines significantly reduce the work that goes into designing a main program. The main program can be viewed as a general controller delegating detailed tasks to subroutines. This lets the programmer concentrate more on the total job to be done when designing the main program. This technique also uses less memory, since each subroutine exists only once in memory and not many times. However, the use of subroutines slows down the exectution speed of a program slightly because of the extra work involved in pushing and pulling registers to and from a stack. You must balance a group of conflicting requirements; memory use, execution speed, and programming ease. In the real world, the most important item is whether the final program works well, and programming ease certainly helps attain this.

**Main Program Responsibilities**

Before a subroutine is called the main program must establish the S stack. The S stack is an unused area of memory in which to store data. To use it as the S stack, the S register must be loaded with the highest address of that area plus one. Data can now be pushed into the stack and later pulled out. The S stack can be established at a memory address higher than the program, seen in Fig. 7-3.

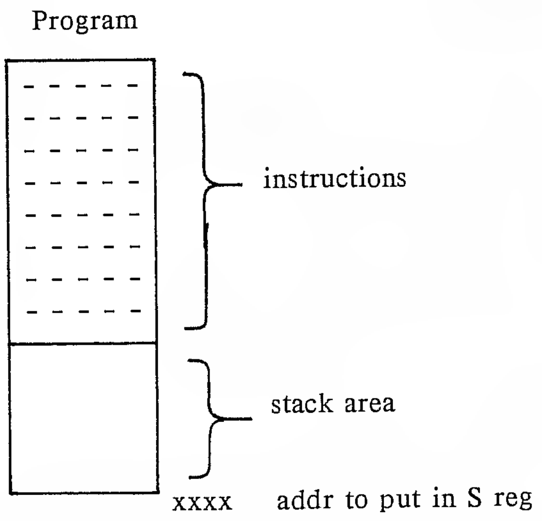


Fig. 7-3 Establishing the S Stack.

A byte is pushed onto the stack by first decrementing the contents of the S register by one and then storing the byte in the memory address specified by the S register. This process is automatically performed by the JSR, BSR, and PSHS instructions. Look at Fig. 7-3: as more bytes are pushed onto the stack, the S register is decremented further and the bytes are stored in memory locations closer and closer to the program. The correct size of the reserved area must be calculated. If the area is too small, the stack will eventually grow into the program instruction area and destroy the program. The required size of the stack is determined by three characteristics of the total program.

One characteristic is how many registers will be stored in the stack when a subroutine is called. The minimum is one register, the PC register, which requires two bytes. However, the subroutine also stores the contents of registers before it uses them. This is so the registers can be restored to their original state before returning to the main program. The number of registers that could be stored by a subroutine vary from none to all except PC and S. Therefore, the stack must be from two bytes long (just PC stored) to 12 bytes long (all possible registers stored) before calling a subroutine.

Another characteristic that determines the size of the stack is the maximum number of levels of subroutines there are in the program. The number of levels is the number of subroutines called before returning to the main program. This number increases as one subroutine calls another subroutine. Fig. 7-4 depicts a program with two levels of subroutines.

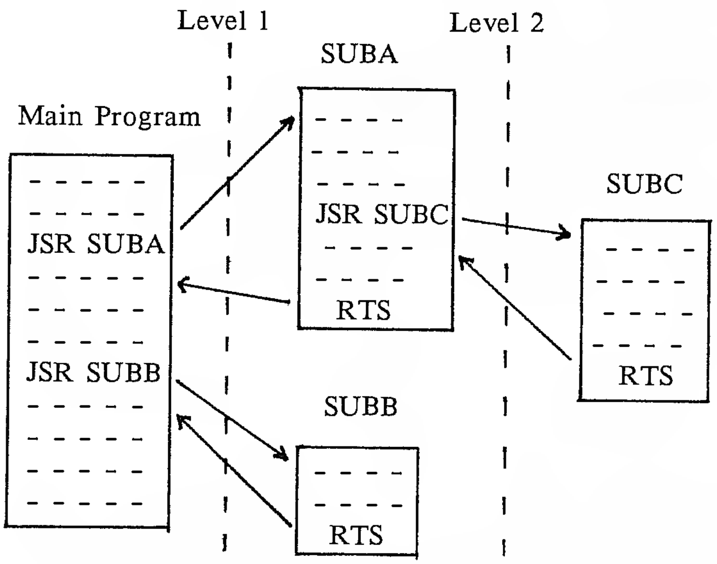


Fig. 7-4 Multiple Levels of Subroutines.

In Fig. 7-4, the main program would have to establish a stack big enough to also hold the registers saved when SUBA is called and when SUBA calls SUBC. A subroutine such as SUBC called by another subroutine is said to be **nested**. This method, where the main program establishes one stack for all the subroutines to use, results in the fastest program execution, but requires more planning by the programmer. Alternatively, SUBA could assume all the responsibilities of a program calling a subroutine and establish its own S stack to be used when calling SUBC. This makes planning the stack sizes easier, since each section would only be concerned with subroutines it calls directly. However, this requires more instructions in SUBA and reduces execution speed. This is another example of trading programming ease for execution speed.

The final characteristic that affects the size of the S stack is whether the main program or any subroutines will use it for other purposes, such as temporary storage data or for data manipulation. The number of bytes required for these other purposes will have to be added to the number determined by the preceding considerations to arrive at the total S stack size.

Another responsibility of a main program or any program calling a subroutine is passing the argument, or data, to the called subroutine and receiving the results. Some subroutines do not require an argument, such as a subroutine that clears the screen. The calling program simply calls it; when that subroutine clears the screen it returns to the calling program with no data to give it. The source code of a sample subroutine to clear the screen is in Listing 7-4.

```
1000 *SUBROUTINE NAME: CSCREN
1010 *BY AUTHOR - DATE
1020 *THIS SUBROUTINE WILL CLEAR THE TEXT
1030 *SCREEN. IT USES 4 BYTES OF THE S STACK.
1040 ****************************************
1050 CSCREN  PSHS A,X,CC     SAVE A,X,CC REGS
1060         LDX #$400       START OF DISPL BUF
1070         LDA #$60        CODE OF SPACE
1080 CS1     STA ,X+         PUT IN BUFFER
1090         CMPX #$5FF      END OF BUFFER?
1100         BLS CS1         IF NOT,DO AGAIN
1110         PULS A,X,CC     RESTORE REGS
1120         RTS             RETURN
1130 ****************************************
```

Listing 7-4 THE CSCREN Subroutine.

Line 1030 provides an important piece of information: how many stack bytes the subroutine will use. This amount is in addition to the two bytes used to store the PC register. The source code of this subroutine, and others, is started at high line numbers to allow the main program's source code to be put in low numbered lines starting at 100. Line 1050 stores the

A, X, and CC registers in the stack because this subroutine uses them. At line 1110, the A, X, and CC registers are pulled from the stack, restoring them to their original contents. This lets the main program continue with all its registers unmodified.

```
1200 *SUBROUTINE NAME: VIDEO
1210 *BY AUTHOR - DATE
1220 *THIS WILL CALCULATE THE VIDEO DISPLAY CODE
1230 *OF A CHARACTER WHOSE ASCII CODE IS IN A. THE
1240 *CHARACTER SET IS UPPER CASE ALPHANUMERIC AND
1250 *SYMBOLS. THE VIDEO CODE WILL BE RETURNED IN A.
1260 *THIS SUBROUTINE USES 1 BYTE OF THE S STACK.
1270 ************************************************
1280 VIDEO    PSHS CC          SAVE REGS
1290          CMPA #$40        CHAR TYPE?
1300          BHS VIS          IF NO CNV, GOTO
1310          ADDA #$40        CNV TO VIDEO
1320 VIS      PULS CC          RESTORE REGS
1330          RTS              RETURN
1340 ************************************************
```

Listing 7-5 The VIDEO Subroutine.

If one or two bytes of data is to be passed to a subroutine, the data is normally put in a specific register by the calling program. When the subroutine is called, it performs its task upon the data in the predesignated register. The result of the subroutine is returned to the calling program in a predesignated register. Listing 7-5 is an example of a subroutine that receives one byte in the A register, manipulates that byte, and returns the result via the same register.

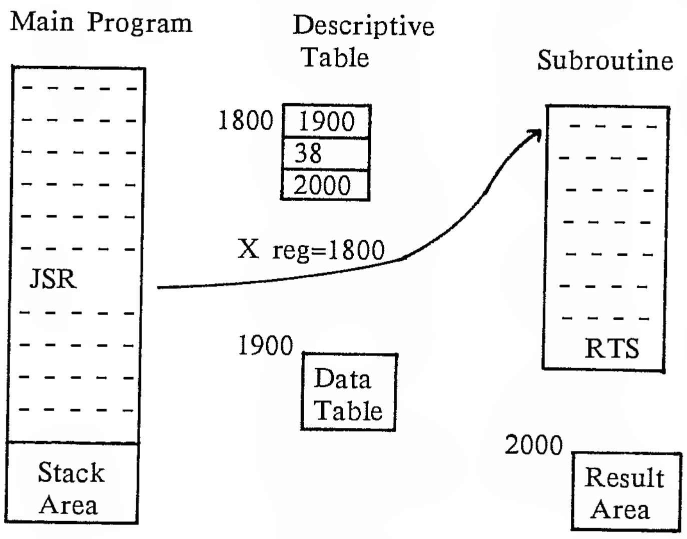


Fig. 7-5 Passing Data to a Subroutine with a Table.

When large amounts of data are to be acted on by a subroutine, it is best to transfer an address to the subroutine. The main or calling program should organize the data in memory as a table or list. Then a table describing that list should be built. The descriptive table should contain the starting address of the data list, the number of elements in the data list, and the destination address (where the subroutine should deposit the results). Then the calling program transfers the address of the descriptive table to the subroutine via a register. This concept is illustrated in Fig. 7-5.

```
1400 *SUBROUTINE NAME: HXVAL
1410 *BY AUTHOR - DATE
1420 *THIS SUBROUTINE WILL CONVERT A STRING OF
1430 *ASCII CODES OF HEX DIGITS TO THEIR HEX VALUE.
1440 *IT WILL RECEIVE THE ADDR OF THE DESCRIPTIVE
1450 *TABLE IN Y. IN THAT TABLE,THE FIRST 2 BYTES ARE
1460 *THE ADDR OF THE NUMERIC TEXT STRING, THE 3RD
1470 *BYTE CONTAINS THE NO OF CHARACTERS, AND THE 4TH
1480 *AND 5TH CONTAIN THE ADDR AT WHICH TO STORE THE
1490 *RESULTING HEX VALUE. IT USES 7 BYTES OF THE STACK.
1500 ***********************************************
1510 HXVAL   PSHS A,B,X,Y,CC SAVE REGS
1520         LDX ,Y++        GET ADDR OF DATA
1530         LDB ,Y+         GET CNT OF DATA
1540         STB HXCNT       STORE COUNT
1550         LDY ,Y++        GET OUTPUT ADDR
1560         CLR ,Y          CLR OUTPUT AREA
1570         BITB #$01       IS CNT ODD?
1580         BEQ HXA         IF NOT, SET EVEN
1590         LDB #$01        SET ODD SWITCH
1600         BRA HXB         CONTINUE
1610 HXA     CLRB            SET EVEN
1620 HXB     LDA ,X+         GET CHAR CODE
1630         CMPA #$39       GREATER THAN 9?
1640         BHI HXC         IF SO,ALPHA CONV
1650         SUBA #$30       CNV TO HEX DIGIT
1660         BRA HXD         CONTINUE
1670 HXC     SUBA #$37       CNV TO HEX DIGIT
1680 HXD     CMPB #$00       EVEN OR ODD?
1690         BEQ HXE         EVEN
1700         ORA ,Y          MERGE HEX DIGITS
1710         STA ,Y+         STORE THEM
1720         CLRB            SET EVEN
1730         BRA HXF         CONTINUE
1740 HXE     LDB #$10        MULTIPLIER
1750         MUL             SHIFT INTO UPPER NIBBLE
1760         STB ,Y          STORE IT
1770         LDB #$01        SET ODD
1780 HXF     DEC HXCNT       DECR CNT
1790         BNE HXB         IF NOT 0,DO AGAIN
1800         PULS A,B,X,Y,CC RESTORE REGS
1810         RTS             RETURN
1820 ************************************************
1830 HXCNT   RMB 1           CNT OF BYTES
1840 ************************************************
```

Listing 7-6 The HEXVAL Subroutine.

The address, $1800, of the descriptive table in Fig. 7-5 is passed to the subroutine in the X register. The subroutine works on the 38 elements of data in the table at address $1900 and puts the results in memory starting at address $2000. The calling program, through the descriptive table, completely controls the subroutine. The calling program is the master and the subroutine is the slave. An example of a subroutine controlled with a descriptive table can be seen in Listing 7-6.

The subroutine performs a task similar to that of BASIC's VAL command. It generates the positive binary value of a string of text hexadecimal digits. For example, the hexadecimal value of $0A5C is generated given the text string "A5C" in ASCII code. The resulting value is right justified. That is the result is $0A5C and not $A5C0, two very different answers.

To use the HXVAL subroutine, the main or calling program builds a table in memory to control the subroutine. If a text string is in memory at address $1800 such as:

```
1800   31 32 46 38          (text = 12F8)
```

then a table would be constructed at address $1700, for example:

```
1700   18 00         (addr of text string)
1702   04             (no. of characters)
1703   21 20                 (output addr)
```

Just before the subroutine is called, the X register is loaded with the address ($1700) of the above table:

```
LDX #$1700
JSR HXVAL
```

LDX loads the table address and JSR calls the HXVAL subroutine. After the subroutine completes, the main program execution resumes with the instruction immediately after the JSR. The subroutine HXVAL would have generated and stored the hexadecimal value in memory as:

```
2120   12 F8
```

where the main program can use it or direct another subroutine to use it.

Another way to transfer many bytes to a subroutine is in a stack, preferably the U stack. In this technique the U stack is established and data put in it by the main or calling program. A descriptive table is built indicating the number of items in the stack and where to put the results. The address of the descriptive table is passed to the subroutine in the X or Y register and the U register points to the top of the U stack. Use the U stack instead of the S stack because the S stack is used to store registers. The S stack can be used to pass data, but be very careful about keeping track of where the data and registers are in it.

In summary, the main or calling program has the following responsibilities when using subroutines:

1) establish an S stack of appropriate size.
2) pass data to the subroutine.
   A) in a register. B) in a table. C) in a stack.

**Subroutine Responsibilities**

A subroutine has responsibilities that must be done before it can work properly with the calling program. Let's look at these responsibilities in the order a subroutine would perform them.

The first responsibility is to save the contents of any registers it will use and therefore modify in the S stack. This lets the subroutine restore the registers to their original contents just before returning to the calling program. However, not all the registers need or can be saved. The PC register does not have to be saved because the JSR has already done that. The S register can not be saved because a stack pointer can not be saved in its own stack. Therefore, the only registers that may need to be saved are A, B, D, X, Y, U, CC, and DP. Of these, the registers that should be saved are those the subroutine uses. For example, the sample subroutine CSCREN saves the A, X, and CC registers at line 1050. The A and X registers are explicitly used in this subroutine and the CC register is implicitly used.

Subroutines that return their result via a register to the calling program need not save that register. For example, the sample subroutine VIDEO does not save the A register, at line 1280, because its final contents will be the result to return to the calling program and not its original contents.

One may adopt a programming discipline where the CC register does not have to be saved in a subroutine. This speeds up its execution by a few microseconds and requires one less byte of stack area for each subroutine. The technique requires that the state of the bits in the CC register be tested, or branched on, immediately after they are set or cleared. In other words, a branch instruction should immediately follow an instruction whose results are to be tested. Therefore, a subroutine does not have to preserve the CC register for later use by the calling program. For example, in the following source code:

```
DEC TCNT
BNE TA1
JSR FIX
STA ,Y+
```

the subroutine FIX does not have to save the CC register, which indicates the result of the DEC instruction, because its state has already been tested by the BNE instruction.

A case where the CC register does not have to be saved is when the subroutine passes its results back in the CC register. For example, a subroutine that searches an area of memory for a particular value may indicate a find or no-find result in the CC register.

A subroutine may not need to save any registers if the main program doesn't require its registers to be the same after the subroutine returns. In this case the main program must be designed to use memory instead or registers to hold data at the time a subroutine is called. This arrangement simplfies subroutine design and reduces stack requirements, but makes the main program somewhat more complicated.

The second responsibility goes hand-in-hand with the first; restoring the contents of the saved registers. The registers pushed into the S stack at the beginning of the subroutine must be pulled at the end of the subroutine. The same number of bytes must be pushed then pulled so the S register contains its original value; this lets the RTS instruction properly pull the PC register from the stack. The calling program may also depend on the S register containing its original value.

If a subroutine establishes its own S stack for use in calling another subroutine, or any other use, the subroutine should save the contents of the S register in a pair of reserved memory locations after saving the other registers to be used. The subroutine can then set up an S stack for its own uses. On completion of the subroutine, the S register must be restored by loading it with its saved value. Then the other registers can be restored by pulling them from the S stack.

The final duty of a subroutine is to return to the calling program. The RTS pulls the original contents of PC from the S stack and puts it in the PC register, causing program execution to resume at the instruction immediately following the JSR. Another way to return is to combine the PULS (restore registers from the S stack) and RTS instructions into one: PULS x,x,x,PC. This one instruction restores registers and pulls the PC register from the S stack.

A skeleton form of a subroutine that does not establish its own S stack is shown in Fig. 7-6.

```
NAME    PSHS x,x,...        save registers
        - - - - -
        - - - - -      }
        - - - - -      }    task instructions
        - - - - -      }
        - - - - -
        PULS x,x,...        restore same registers
        RTS                 return to calling prog
```

Fig. 7-6 Skeleton of a Simple Subroutine.

The responsibilities of the subroutine in Fig. 7-6 can be summarized as:
1) save registers to be used.
2) perform task.
3) restore same registers.
4) return to calling program.

The skeleton of a subroutine that establishes its own S stack is shown in Fig. 7-7.

```
NAME     PSHS x,x,...          save registers
         STS SAVES             save old S contents
         LDS #YYY+NEWS         establish new stack
         - - - - -
         - - - - -
         - - - - -
         - - - - -             task instructions
         - - - - -
         - - - - -
         LDS SAVES               restore S
         PULS x,x,...          restore same registers
         RTS                   return to calling prog
SAVES    RMB 2                 area to store S
NEWS     RMB YYY               new stack area
```

Fig. 7-7 Skeleton of Subroutine with Its Own S Stack.

The responsibilities of the subroutine in Fig. 7-7 can be summarized as:
1) save registers to be used.
2) save S register.
3) establish new S stack.
4) perform task.
5) restore S register.
6) restore used registers.
7) return to calling program.

**Tying It Together**

Each time a subroutine is called, more registers are pushed on the S stack. Each time a subroutine returns, the same registers are pulled from the stack. As a result the value in the S register after returning is the same as it was just before calling the subroutine. The value in the S register is the same before and after a JSR or BSR instruction.

A specific example of a main program using subroutines is shown in Listing 7-7. Lines 100 - 310 make up the main program. This program clears the screen with the CSCREN subroutine and then displays a line of text. The video display codes are generated with the VIDEO subroutine. Upon completion it stays in a loop at line 270.

```
100 *PROGRAM NAME: SAMPLE
 110 *BY AUTHOR - DATE
 120 *THIS PROGRAM WILL USE THE CSCREN TO CLEAR THE
 130 *SCREEN, THEN DISPLAY A LINE OF TEXT USING THE
 140 *VIDEO SUBROUTINE. UPON COMPLETION IT WILL
 150 *STAY IN A LOOP. PRESS RESET TO EXIT THIS PROGRAM.
 160 ***************************************************
 170 SAMPLE  LDS #6+NSTAK    ESTABLISH STACK
 180         JSR CSCREN      CLR SCREEN
 190         LDX #SATEXT     GET ADDR OF TEXT
 200         LDY #$480       ADDR OF DISPLAY LINE
 210 SABG    LDA ,X+         GET TEXT CHAR
 220         CMPA #$04       END OF TEXT CODE?
 230         BEQ SAEND       IF SO, GOTO END
 240         JSR VIDEO       CNV TO VIDEO
 250         STA ,Y+         PUT ON SCREEN
 260         BRA SABG        DO AGAIN
 270 SAEND   BRA SAEND       WAIT IN LOOP
 280 **************************************************
 290 NSTAK   RMB 6           STACK AREA
 300 SATEXT  FCC !THIS TEXT DISPLAYED WITH TWO SUBROUTINES!
 310         FCB 04          END OF TEXT CODE
 320 **************************************************
1000 *SUBROUTINE NAME: CSCREN
1010 *BY AUTHOR - DATE
1020 *THIS SUBROUTINE WILL CLEAR THE TEXT SCREEN
1030 *IT REQUIRES 4 BYTES OF THE S STACK
1040 **************************************************
1050 CSCREN  PSHS A,X,CC     SAVE A, X, CC
1060         LDX #$400       START OF DISPL BUF
1070         LDA #$60        SPACE CODE
1080 CS1     STA ,X+         PUT IN BUFFER
1090         CMPX #$5FF      END OF BUFFER?
1100         BLS CS1         IF NOT,DO AGAIN
1110         PULS A,X,CC     RESTORE A,X, CC
1120         RTS             RETURN
1130 **************************************************
1200 *SUBROUTINE NAME: VIDEO
1210 *BY AUTHOR - DATE
1220 *THIS WILL CALCULATE THE VIDEO DISPLAY CODE
1230 *OF A CHARACTER WHOSE ASCII CODE IS IN A. THE
1240 *CHARACTER SET IS UPPER CASE ALPHANUMERIC AND
1250 *SYMBOLS. THE VIDEO CODE IS RETURNED IN A.
1260 *THIS SUBROUTINE USES 1 BYTE OF THE S STACK.
1270 **************************************************
1280 VIDEO   PSHS CC         SAVE REGS
1290         CMPA #$40       CHAR TYPE?
1300         BHS VIS         IF NO CNV,GOTO
1310         ADDA #$40       CNV TO VIDEO
1320 VIS     PULS CC         RESTORE REGS
1330         RTS             RETURN
1340 **************************************************
1350         END
```

Listing 7-7 The SAMPLE program.

The stack area is established at line 290 and the S register is loaded at line 170 with the address of the last location of the stack area plus one. The stack area size is determined by the subroutine that stores the largest number of bytes. In this case the stack area is six bytes long - two to store the PC register plus four needed by the CSCREN subroutine.

Listing 7-7 is assembled with the A/IM/WE command without any errors. To run it, go to ZBUG and enter the command GSAMPLE. The screen should be cleared and the text line defined at line 300 displayed. To return to ZBUG, press the Reset button on the back of the Color Computer.

A subroutine can assume the responsibility of establishing its own S stack for use by a subroutine it calls. A calling program establishes a stack for the subroutine it calls to store its registers in. This relationship is illustrated in Fig. 7-8.

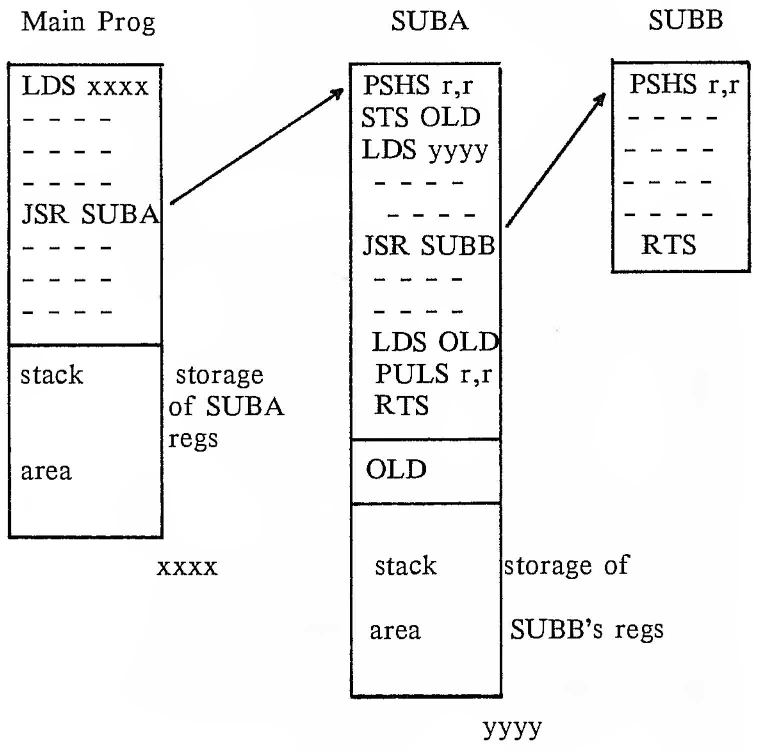


Fig. 7-8 A Subroutine Establishing Its Own S Stack.

The Color Computer can be set up to perform **multi-tasking** or **multi-programming**, the process of executing more than one program on a time-sharing basis. Two or more programs would be in memory at the same time, and a supervisor program would direct the MPU to execute one program for a time, and then execute another program for a time. Memory can be saved by having just one set of currently used subroutines in memory for any program to use. However, a problem arises if a program is temporarily suspended while it is executing a shared subroutine. If that subroutine has its own data area for manipulating data, data may be there when it is suspended. If another program uses that same subroutine, the data in the subroutine's data area will be modified. When the first program is resumed, that subroutine will resume without the original data. This problem is solved by making the subroutines **re-entrant** and having a supervisor that saves and restores registers when switching from one program to another. A re-entrant subroutine does not establish its own data areas but manipulates data only within the registers. The CSCREN and VIDEO subroutines are re-entrant, but HXVAL is not.

## INTERRUPTS

An interrupt performs a function similar to that of the JSR or BSR instructions. It causes the MPU to jump to a program section at a specific address. The two categories of interrupts are software and hardware. Software interrupts are caused by executing an SWI, SWI2, or SWI3 instruction. Therefore, they are planned or under the direct control of the program. The hardware interrupts, IRQ, FIRQ, and NMI, are initiated by an external electrical device that activates a signal to the appropriate pins of the MC6809E dual-in-line package. They might be more accurately called external interrupts because they come from outside the MPU. A hardware interrupt can occur at almost any time and so be considered unplanned; the moment of its occurrence can not be controlled by the program.

When an interrupt is processed, the MPU is directed to a program section known as an **interrupt handler** by the interrupt vector table in ROM. The interrupt vector table resides in ROM at addresses $FFF2 - $FFFF. This table contains seven addresses, each occupying two consecutive locations. Each of six is dedicated to directing the MPU when one of six interrupts occurs. The seventh address directs the MPU when a RESET occurs.

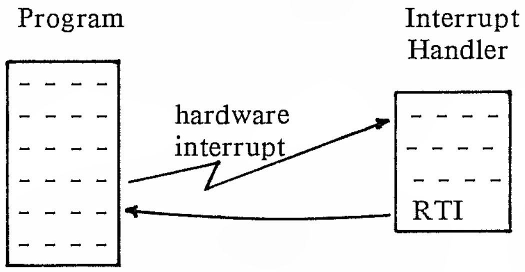


Fig. 7-9 General Concept of a Program Being Interrupted.

The interrupt handler is the program section the MPU is directed to by an interrupt via the interrupt vector table. The interrupt handler may have a task to perform, and upon completion execute an RTI instruction which will cause the interrupted program to resume its operation. A program that is interrupted may never be returned to, however. For example, the interrupt may signal a condition that tells the program to quit doing its present task and work on another. Fig. 7-9 shows a program being interrupted by a hardware interrupt and the interrupt handler returning to the program. This looks similar to a program calling a subroutine.

**Interrupt Operation**

The operation of the various interrupts will be presented. The RESET will also be covered since its operation is similar to that of the hardware interrupts. (For more information see Chapters 3 and 5.) The interrupt vector table, which directs the MPU upon an interrupt, is shown in Table 7-1.

| User | Address | Effective Address |
|---|---|---|
| RESET | FFFE + FFFF | $A027 |
| NMI | FFFC + FFFD | $0109 |
| SWI | FFFA + FFFB | $0106 |
| IRQ | FFF8 + FFF9 | $010C |
| FIRQ | FFF6 + FFF7 | $010F |
| SWI2 | FFF4 + FFF5 | $0103 |
| SWI3 | FFF2 + FFF3 | $0100 |

Table 7-1 Interrupt Vector Table.

To use the interrupts, a vector jump to each interrupt handler must be stored at each effective vector address in RAM. When the Color Computer is turned on, the BASIC program in ROM performs certain initializing functions, one of which is to store the vector jumps for the interrupts it uses. BASIC sets up the vector jumps for the IRQ and FIRQ interrupts since they are the only ones that it uses. If the EDTASM+ module is also plugged in, the vector jump for the SWI is also set up. The operands of the vector jumps point to their respective interrupt handlers. The complete vector jump table, as organized in RAM, can be seen in Table 7-2 in the mnemonic display format.

| Address | Instruction | Content |
|---|---|---|
| $0100 | JMP xxxx | Jump to SWI3 interrupt handler |
| $0103 | JMP xxxx | Jump to SWI2 interrupt handler |
| $0106 | JMP xxxx | Jump to SWI interrupt handler |
| $0109 | JMP xxxx | Jump to NMI interrupt handler |
| $010C | JMP xxxx | Jump to IRQ interrupt handler |
| $010F | JMP xxxx | Jump to FIRQ interrupt handler |

Table 7-2 Vector Jump Instruction Table.

An interrupt will vector the MPU to one of the vector jump instructions which directs the MPU to the actual interrupt handler. Fig. 7-10 illustrates this path taken by the MPU in response to an IRQ interrupt. Since the vector jump instructions are in RAM they can be changed to point to interrupt handlers written in assembly language.

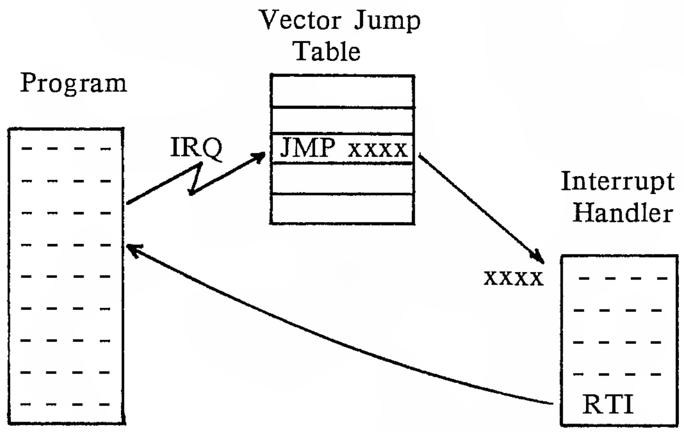


Fig. 7-10 Use of the Vector Jump Table.

The software interrupts are the SWI, SWI2, and SWI3 instructions. They can be used in a fashion similar to the JSR or BSR instructions, but their results are somewhat different.

The SWI instruction sets the E bit of the CC register and causes the P, U, Y, X, DP, B, A, and CC registers to be pushed into the S stack when executed. Then the I and F bits of the CC register are set to mask out any IRQ or FIRQ interrupts that may occur while executing the SWI interrupt handler. The MPU finally executes the vector jump instruction at address $0106, which directs it to the SWI interrupt handler.

The SWI2 instruction sets the E bit of the CC register and causes the P, U, Y, X, DP, B, A, and CC registers to be pushed into the S stack. Then the MPU executes the vector jump instruction at address $0103, which directs it to the SWI2 interrupt handler.

The SWI3 instruction sets the E bit of the CC register and causes the P, U, Y, X, DP, B, A, and CC registers to be pushed into the S stack. Then the MPU will execute the vector jump instruction at address $0100, which directs it to the SWI3 interrupt handler.

The hardware interrupts perform operations similar to the software interrupts. The differences are that the IRQ and FIRQ interrupts can be masked out, or ignored, and the hardware interrupts are initiated by an external device. The I bit set causes the MPU to ignore an IRQ interrupt signal and the F bit set masks out the FIRQ interrupt.

The IRQ interrupt sequence takes place if the I bit is clear and pin 3 of the MC6809E dual-in-line package is temporarily set low. The sequence starts upon completion of the currently executing instruction. The IRQ interrupt sequence consists of first setting the E bit in the CC register, then pushing the P, U, Y, X, DP, B, A, and CC registers into the S stack. Then the I bit is set to mask out any IRQ interrupts that may occur while executing the IRQ interrupt handler. Finally, the MPU executes the vector jump at address $010C, which directs the MPU to the IRQ interrupt handler. This sequence takes about 20 microseconds on the Color Computer.

The FIRQ interrupt sequence takes place if the F bit is clear and pin 4 of the MC6809E dual-in-line package is temporarily set low. The sequence starts upon completion of the currently executing instruction. The FIRQ interrupt sequence consists of first clearing the E bit in the CC register, then pushing the PC and CC registers into the S stack. Then the I and F bits are set to mask out any IRQ or FIRQ interrupts that may occur while executing the FIRQ interrupt handler. Finally, the MPU executes the vector jump at address $010F, which directs the MPU to the FIRQ interrupt handler. This sequence takes about 10 microseconds, much faster than any of the other hardware interrupts. This interrupt is used when the program must respond very quickly to an external stimulus.

The NMI interrupt sequence is inhibited after a RESET until the S register has been loaded. The sequence is initiated by pin 2 of the MC6809E dual-in-line package temporarily set low, and starts upon completion of the currently executing instruction. The interrupt sequence consists of first setting the E bit in the CC register, then pushing the P, U, Y, X, DP, B, A, and CC registers into the S stack. Then the I and F bits are set to mask out any IRQ or FIRQ interrupts that may occur while executing the NMI interrupt handler. Finally, the MPU executes the vector jump at address $0109, which directs the MPU to the NMI interrupt handler. This sequence takes about 20 microseconds on the Color Computer.

The RESET sequence is started when the Color Computer is turned on, or the Reset button is pressed. The I and F bits are set, the DP register is cleared, and the MPU is directed to address $A027 by the interrupt vector table. The RESET sequence does not use a vector jump, but instead sends the MPU directly to BASIC ROM.

**Pre-Interrupt Responsibilities**

Several operations must be performed before using an interrupt within an assembly language program. They are presented in the order in which they should be performed.

The first instruction of an assembly language program that uses interrupts should be ORCC #$50. This sets the I and F bits of the CC register, masking out any IRQ and FIRQ interrupts until the program is ready for them.

Next the vector jump instruction for each type of interrupt that will be used must be put in memory at the effective vector address. The jump operand is the absolute address of the interrupt handler for that interrupt type. For example, if one were going to use the SWI2 software interrupt, the vector jump should be put in memory at address $0103 as 7E xx xx. $7E is the JMP op code and xxxx is the absolute address of the SWI2 interrupt handler. The interrupt handler should exist in memory along with the main program and its subroutines. It is a programmer's responsibility to have assembled the interrupt handler and to know where it resides in memory.

An S stack must be established because all the interrupts cause MPU registers to be pushed into it. Upon returning from the interrupt handler, the registers are pulled from the stack by the RTI so the interrupted program can resume its operation. The size of the stack area is determined by several considerations. The stack must be large enough for any subroutine use. The stack area must also be large enough for the anticipated interrupts. The FIRQ interrupt requires only three bytes of stack area since it causes only the PC and CC registers to be pushed into the stack. All the other interrupts require 12 bytes; they push all the registers except S. If a subroutine is allowed to be interrupted, the stack area must be that required by the subroutine plus that required by the interrupt. This is similar to nested subroutines.

Another factor is how many interrupts will be processed before returning to the interrupted program. It is possible for an interrupt handler to be interrupted by an external or software interrupt, in which case more registers will be pushed into the stack before the previous set has been pulled from it. Therefore, each interrupt processed before returning will require more stack area. The stack area must be enlarged by three bytes for an FIRQ, or 12 bytes for any other interrupt, for each interrupt that occurs before returning. The FIRQ, NMI, and SWI interrupts set the I and F bits so their interrupt handlers will not be interrupted by an FIRQ, NMI, or IRQ. The IRQ interrupt sets the I bit so its interrupt handler will not be interrupted by another IRQ.

At this point the program is ready to use any interrupt except for the IRQ and FIRQ. IRQ and FIRQ are generated by other electronic devices within the Color Computer that must be initiated by the program before they will be generated. These electronic devices are described in Chapter 9. After one or both are initialized, the I and/or F bits should be cleared to allow them. To summarize, before using interrupts within an assembled program:

1) mask interrupts until ready (set I and F bits).
2) set up vector jump instructions.
3) establish S stack.
4) if using an IRQ or FIRQ interrupt.
    A) initialize their source.
    B) clear the I and/or F bits.

These responsibilities are illustrated in Fig. 7-11 as a skeleton program preparing to use the SWI2 interrupt.

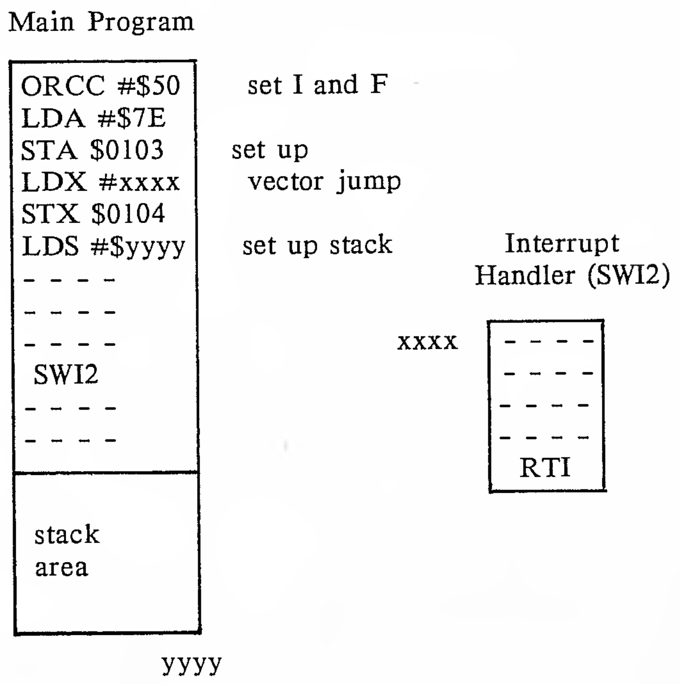


Fig. 7-11 Preparing for SWI2 Interrupt.

**Interrupt Handlers**

The interrupt handler program section has its duties to perform if it is to work properly with the main program. The duties are presented in the order they should be performd.

None of the interrupt handlers, except FIRQ, need store MPU registers; the interrupt sequence will already have done that. The FIRQ handler may save the registers it uses if the interrupted program requires it.

Now the interrupt handler can perform its task. That may be any specific task, as a subroutine has its task, or it may direct the MPU to start executing a different program.

The interrupt handler for the IRQ and FIRQ should reinitialize the device that generates the external interrupts so they will occur again. (This is described in Chapter 9.) If the FIRQ interrupt handler has pushed some registers at its beginning, the registers must be pulled from the S stack.

The last instruction of an interrupt handler is RTI, which returns the MPU to the interrupted program. The RTI will pull the CC register from the S stack and inspect the E bit. If the E bit is set, all the MPU registers except S will be pulled from the stack. If the E bit is clear, only the P register will be pulled from the stack. If there was any modification of the S register by the interrupt handler, the S register should be restored to its original contents before executing the RTI, so the RTI will work properly. An example of a skeleton SWI2 interrupt handler can be seen in Fig. 7-11.

The responsibilities of an interrupt handler that will return to the interrupted program can be summarized:

1) if FIRQ, store registers if necesary.
2) perform task.
3) if IRQ or FIRQ, re-initilize interrupt source.
4) if FIRQ, restore saved registers.
5) return with an RTI.

**Wrap Up**

The use of the IRQ interrupt in conjunction with subroutines is demonstrated in Listing 7-8, named SAMPL1. It is a modified version of the program SAMPLE. A block diagram of SAMPL1 can be seen in Fig. 7-12.

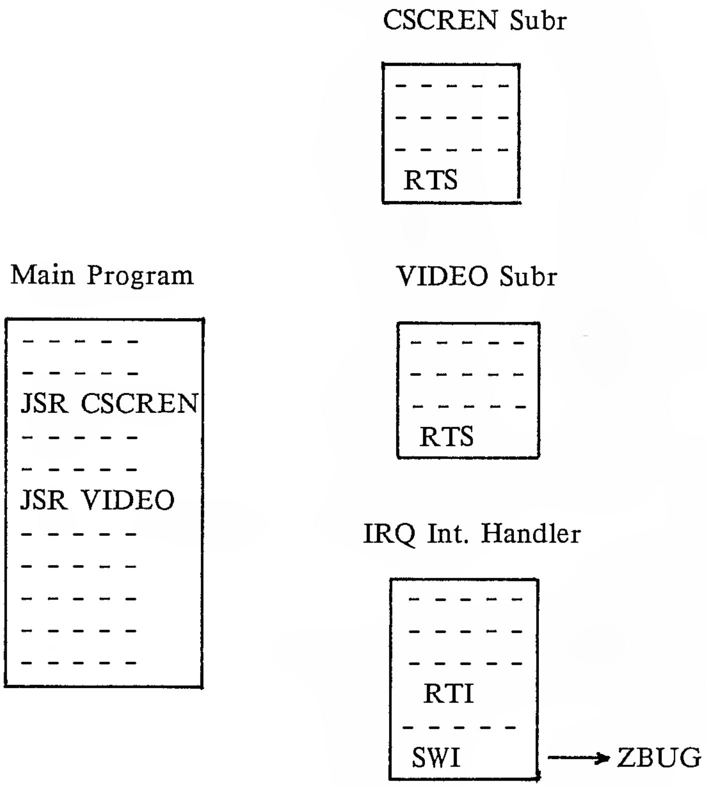


Fig. 7-12 Block Diagram of SAMPL1 Program.

SAMPL1 uses two subroutines, CSCREN and VIDEO, to clear the screen and display a line of text. It then waits for an IRQ interrupt. The IRQ interrupt handler shifts whatever is on the screen one position to the left on every tenth occurrence of the IRQ interrupt. In the Color Computer, the IRQ can be set up to occur every 16.67 milliseconds, or 60 times a second, as in this program. The interrupt sources are described in detail in Chapter 9. The IRQ interrupt handler, upon its 2000th exectuion, returns to ZBUG with an SWI interrupt. This program runs for 33 seconds.

The S stack area is determined by the user or users that use the largest amount of stack area. In this program the maximum size is 12 bytes for the IRQ interrupt plus another 12 bytes for the SWI, a total of 24 bytes.

```
 100 *PROGRAM NAME: SAMPL1
 110 *BY AUTHOR - DATE
 120 *THIS PROGRAM WILL USE THE CSCREN TO CLEAR THE
 130 *SCREEN, THEN DISPLAY A LINE OF TEXT USING THE
 140 *VIDEO SUBROUTINE. UPON COMPLETION IT WILL
 150 *STAY IN A LOOP WAITING FOR AN IRQ INTERRUPT.
 160 ***********************************************
 170 SAMPL1  ORCC #$50       SET I AND F BITS
 180         LDA #$7E        JMP OP CODE
 190         STA $010C       STORE IT
 200         LDY #LSHIFT     ADDR OF INT HANDLER
 210         STY $010D       STORE IT
 220         LDS #24+NSTAK   ESTABLISH STACK
 230         JSR CSCREN      CLEAR SCREEN
 240         LDX #SATEXT     GET ADDR OF TEXT
 250         LDY #$480       ADDR OF DISPL LINE
 260 SABG    LDA ,X+         GET TEXT CHAR
 270         CMPA #$04       END OF TEXT CODE?
 280         BEQ SAEND       IF SO,GOTO END
 290         JSR VIDEO       CNV TO VIDEO
 300         STA ,Y+         PUT ON SCREEN
 310         BRA SABG        DO AGAIN
 320 SAEND   LDA $FF03       READ CRB
 325         ORA #$05        SET BITS 0+2
 330         STA $FF03       STORE IN CRB
 340         LDA $FF02       READ DRB(CLR FLAG)
 350         ANDCC #$EF      CLR I BIT
 360 SALOOP  BRA SALOOP      LOOP AND WAIT
 370 ***********************************************
 380 NSTAK   RMB 24          STACK AREA
 390 SATEXT  FCC !THIS TEXT DISPLAYED WITH TWO SUBROUTINES!
 400         FCB 04          END OF TEXT CODE
 410 ***********************************************
1000 *SUBROUTINE NAME: CSCREN
1010 *BY AUTHOR - DATE
1020 *THIS SUBROUTINE WILL CLEAR THE TEXT SCREEN
1030 *IT REQUIRES 3 BYTES OF THE S STACK
1040 ***********************************************
1050 CSCREN  PSHS A,X        SAVE A AND X
1060         LDX #$400       START OF DISPL BUF
1070         LDA #$60        SPACE CODE
1080 CS1     STA ,X+         PUT IN BUFFER
1090         CMPX #$5FF      END OF BUFFER?
```

```
1100         BLS CS1        IF NOT,DO AGAIN
1110         PULS A,X       RESTORE A AND X
1120         RTS            RETURN
1130 **********************************************
1200 *SUBROUTINE NAME: VIDEO
1210 *BY AUTHOR - DATE
1220 *THIS WILL CALCULATE THE VIDEO DISPLAY CODE
1230 *OF A CHARACTER WHOSE ASCII CODE IS IN A. THE
1240 *CHARACTER SET IS UPPER CASE ALPHANUMERIC AND
1250 *SYMBOLS. THE VIDEO CODE WILL BE RETURNED IN A.
1260 **********************************************
1270 VIDEO   CMPA #$40      CHAR TYPE?
1280         BHS VIS        IF NO CNV,GO TO
1290         ADDA #$40      CNV TO VIDEO
1300 VIS     RTS            RETURN
1310 **********************************************
1400 *IRQ INTERRUPT HANDLER NAME: LSHIFT
1410 *BY AUTHOR - DATE
1420 *THIS INT HANDLER WILL SHIFT WHATEVER IS ON THE
1430 *TEXT SCREEN ONE POSITION TO THE LEFT ON EVERY
1440 *10TH INTERRUPT. ON THE 2000TH INTERRUPT IT WILL
1450 *RETURN TO ZBUG VIA SWI. 24 BYTES OF STACK ARE USED.
1460 **********************************************
1470 LSHIFT  LDX LSCNTA     GET A CNT
1480         LEAX 1,X       INC A CNT
1490         STX LSCNTA     STORE A CNT
1500         CMPX #2000     LAST INTERRUPT?
1510         BEQ LSEND      IF SO,RET TO ZBUG
1520         LDA LSCNTB     GET B CNT
1530         INCA           INC B CNT
1540         STA LSCNTB     STORE B CNT
1550         CMPA #10       TIME TO SHIFT?
1560         BNE LSRET      IF NOT,RETURN
1570         LDX #$401      BUFFER SOURCE
1580         LDY #$400      BUFFER DESTINATION
1590 LSA     LDA ,X+        GET DISPL CODE
1600         STA ,Y+        MOVE DISPL CODE
1610         CMPX #$5FF     END OF BUFFER?
1620         BLS LSA        IF NOT,DO AGAIN
1622         CLRA           GET ZERO
1624         STA LSCNTB     CLR B CNT
1630 LSRET   LDB $FF02      READ DRB(CLR FLAG)
1640         RTI            RETURN TO PROGRAM
1650 LSEND   LDD #$0        CLR REG
1660         STD LSCNTA     CLR A CNT
1670         STA LSCNTB     CLR B CNT
1680         SWI            RETURN TO ZBUG
1690 ********************************************
1700 LSCNTA  FDB 0000       CNT A = 0
1710 LSCNTB  FCB 0          CNT B = 0
1720 ********************************************
1730         END
```

Listing 7-8 The SAMPL1 Program.

In the Listing 7-8 are examples of setting up for subroutine and interrupt use. Assemble the source code into memory with the A/IM/WE command and verify that there are no errors. Then go to ZBUG and run it by typing in GSAMPL1. The vector jump for the IRQ has been modified by this program. Therefore, one should turn the computer off and then on after running and experimenting with the SAMPL1 program, to set that vector jump back to its normal value.

# CHAPTER 8

# Assembly Language and Extended Color BASIC

BASIC, because it exists in internal ROM, is always available and serves as a starting point for many programs, including assembly language programs. Programs assembled on tape can be loaded into memory via BASIC's CLOADM command and executed with the EXEC command. One may also construct subroutines for BASIC programs to use, and assemble them on tape. BASIC can load the subroutines from tape into memory and call them with the USR function. The subroutines can increase execution speed and provide capabilities BASIC does not have.

The BASIC interpreter is composed of many subroutines available for use by assembly language programs. An advantage of these subroutines is that they exist in ROM and therefore do not take up RAM space. Also, since they already exist the time and effort of designing and programming them are avoided.

## USING ASSEMBLED PROGRAMS WITH BASIC

Assembly language programs and subroutines to be used by a BASIC program must share available memory with BASIC. BASIC uses certain areas of memory for its own purposes. One should ensure that assembly language programs to work in concert with a BASIC program do not use BASIC's designated areas in a conflicting manner. It is best to put an assembled program into an area of memory BASIC does not use.

Normally BASIC uses memory locations starting at address 0000 and upward; and starting at the highest RAM address and downward. That is, it uses memory locations starting at both ends of available RAM and works toward the center. This can be seen in Fig. 8-1. These addresses are for a 16K computer, except for those in parentheses, which are for a 32K computer.

| Memory Use | Hexadecimal Address |
|---|---|
| Internal Use | 0000 |
| Text Buffer | 0400 |
| Graphics Buffer | 0600 |
| BASIC Statements | 0601+(N x 600) |
| Numeric Variables | xxxx |
| Unused | 3EDO (7ED0) |
| S Stack | 3F36 (7F36) |
| String Variable Area | 3FFF (7FFF) |

Fig. 8-1 Memory Use by Extended Color BASIC.

Memory locations 0000 - $03FF are used by BASIC for internal operations. Locations $400 - $5FF are for the text display buffer. The graphics buffer is composed of N pages, and each page is $600 locations long. The graphics display buffer is initialized to four pages so it occupies addresses $600 - $1E00. The number of pages in the graphics buffer can be changed with the PCLEAR command.

BASIC program statements are stored in memory after the graphics buffer. Addresses $19 and $1A contain the address at which the statements begin in memory. With a graphics buffer of four pages, the statements are stored starting at address $1E01. This can be verified by seeing that $1E01 is contained in addresses $19 and $1A. Numeric variables are stored after the statements. Addresses $1B and $1C contain this starting address. Memory is not assigned to each numeric variable until it is used by the BASIC program; seven locations are required for each numeric variable and five are required for each element of a numeric array. The unused area in the middle of RAM is the area not used by BASIC programs which do not use all of RAM. The area at the highest address of RAM is reserved for string variables. This area is initialized to 200 decimal bytes but can be changed with the CLEAR command. Below the string variable area is BASIC's S stack. The stack area is about 100 decimal locations long.

**Memory For Assembled Programs**

A goal is to determine the starting and ending addresses of an unused area in which to load assembled programs that will work with BASIC programs. This is not easy to do since the numeric variable area is not established until the BASIC program is run. You may use unused area in the middle of RAM for assembled programs only when the BASIC programs are short and simple. Then it is relatively safe to load an assembled program into memory somewhere between addresses $2600 and $3E00 (16K) or $7E00 (32K) without performing extensive memory use calculations.

Another way to find a memory area for assembled programs is to use the BASIC CLEAR command at the very beginning of a BASIC program or before the program is run. This lets the programmer control two conditions affecting how memory is used by BASIC. The CLEAR command has the following format:

```
CLEAR  XXX,YYYY
```

This command assigns XXX number of bytes for string variable storage and sets the top of RAM available to BASIC at the address YYYY. Memory above YYYY is reserved for other uses such as assembled programs and subroutines. For example: CLEAR 100,&H3000 reserves 100 decimal bytes for string variable storage and tells BASIC that RAM is available only up to address $3000. After using this command a BASIC program's memory use would be similar to that in Fig. 8-1. However, addresses $3001 and up would not be available to BASIC; the string variable and S stack areas would start at $3000 and extend downward. The area above address $3000 is where assembled programs can be put.

**Executing (EXEC) Assembled Programs**

The EXEC command is used to execute assembled programs and can also be used to call assembled subroutines. The format of the EXEC command is:

```
EXEC  expression
```

The expression can be blank, a constant, a numeric variable, or an arithmetic expression. When the EXEC is performed the current contents of PC are pushed into the S stack and the MPU is directed to start executing at the address specified by the expression.

When the expression is blank, as: EXEC, the MPU is directed to the address contained within addresses $9D and $9E. If $9D contains $31 and $9E contains $00, the MPU will be directed to start executing instructions at address $3100. When the expression is a constant, such as: EXEC 12000, the MPU is directed to start a program at decimal address 12000. Prefixing the constant with &H indicates the number is hexadecimal. Examples are:

```
10 X=&H1000
20 EXEC X
20 EXEC X*3
```

Statement 20 will cause the MPU to start executing at $1000; statement 30 will direct the MPU to address $3000.

When the BASIC interpreter is running, it is using its S stack. An executing assembled program or subroutine must take care to preserve and not destroy the contents of that stack and to return the S register to its original contents before returning to BASIC. There are about 30 decimal bytes available for use in BASIC's S stack into which a program can push the MPU registers. Before returning to BASIC the same registers must be pulled from the S stack to return S to its original value.

A program or subroutine that is to return to BASIC does not have to preserve any of the registers except S. This is unlike an assembly language subroutine which must preserve the contents of the registers used. A skeleton of a program to return to BASIC that does not use the S register is shown in Fig. 8-2.

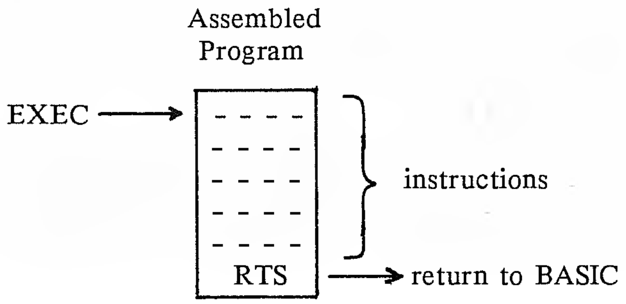
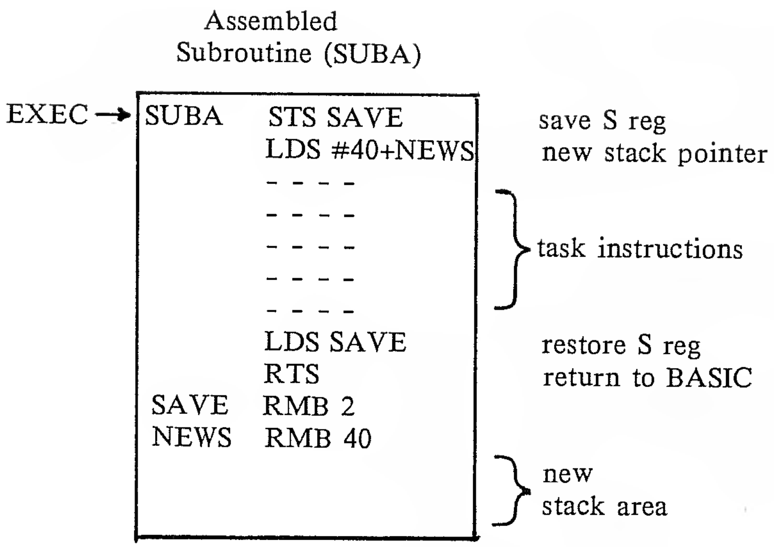


Fig. 8-2 A Subroutine That Does Not Use the S Register.

A program executed from BASIC may establish its own S stack for use by a subroutine it calls. Perhaps the program needs a stack area larger than the 30 bytes available in BASIC's stack; the program should establish its own S stack area. If that program is to later return to BASIC, it must load S with its original value, which should have been saved earlier. Fig. 8-3 shows a skeleton of a subroutine that establishes its own stack. This shows the sequence of a subroutine saving BASIC's S register, setting up its own stack pointer, performing its task, restoring BASIC's stack pointer, and returning to BASIC. Notice that a subroutine called by BASIC does not have to save any MPU registers.

A limitation of the EXEC command is that it can not directly pass an argument to a subroutine. The only way data can be passed to a subroutine is to first put the data in a predesignated memory location. Then the subroutine must always get the data from the same predetermined address. Basically, the EXEC command is not meant to be used to call subroutines, but to simply direct the MPU to a specific program.



```
SUBA    STS SAVE
        LDS #40+NEWS
        - - - -
        - - - -
        - - - -
        - - - -
        - - - -
        LDS SAVE
        RTS
SAVE    RMB 2
NEWS    RMB 40
```

Fig. 8-3 A Subroutine Establishing an S Stack.

An example of using the EXEC command to call an assembled subroutine is presented in Listing 8-1. The BASIC program asks the operator to enter a decimal number. It then displays BASIC's binary floating point representation of that value as it exists in five bytes in memory. The assembled subroutine has the task of displaying the hexadecimal contents of the five bytes on the screen.

```
 10 CLEAR 100,&H2900
 20 CLOADM "PRFP",&H2A00
 30 Y=&H2A00:CLS
 40 INPUT"ENTER A NUMBER  X= ";X
 50 P=VARPTR(X)
 60 PRINT"ITS BINARY FLOATING POINT"
 64 PRINT"REPRESENTATION AT ADDR ";HEX$(P)
 66 PRINT"IN HEX IS"
 68 PRINT
 70 PU=FIX(P/256):PL=P-256*PU
 80 POKE Y,PU:POKE Y+1,PL
 90 EXEC Y+2
100 PRINT"DO YOU WANT TO ENTER ANOTHER"
110 INPUT"NUMBER - Y OR N";N$
120 IF N$="Y" THEN 25
130 END
```

Listing 8-1 The Use of the EXEC Command.

Line 10 reserves memory above address $2900 for use by programs other than BASIC. At line 20 the subroutine object code is loaded into the reserved area. At line 30 the variable Y is set equal to the load address of the subroutine. Then the value for X is obtained, and its pointer is found at line 50. At line 70 is formed the upper and lower bytes of the pointer address which are POKEd into memory (passed to the subroutine) at line 80. Line 90 calls the subroutine, which displays the hexadecimal contents of the five bytes whose address was POKEd into memory. The assembly language subroutine source listing is in Listing 8-2.

```
100 ***********************************************
110 *PROGRAM NAME: PRFP
120 *BY AUTHOR - DATE
130 *THIS PROGRAM WILL RUN AS A SUBROUTINE WHEN
140 *CALLED BY BASIC'S EXEC COMMAND. THE POINTER TO
150 *A NUMERIC VARIABLE SHOULD BE PUT IN ITS FIRST
160 *2 ADDRESSES AND THEN EXEC TO THE THIRD. THE
170 *PROG WILL DISPLAY THE HEX CONTENTS OF THE 5
180 *BYTES AT THE POINTER ADDR. THIS PROG IS RELOCATABLE
190 ***********************************************
200 PRFP    RMB 02          POINTER AREA
210 PR1     LDA #$06        COUNT VALUE
220         STA PRCNT,PCR   PRESET CNT
230         LDY PRFP,PCR    GET DATA POINTER
240         LDU #$484       GET DISPL ADDR
250 PR2     DEC PRCNT,PCR   DEC CNT
260         BEQ PREND       IF 0, GOTO END
270         BRA PR3         GOTO CNV & DISPL
280 PR2A    LEAY 1,Y        INC DATA POINTER
290         LEAU 2,U        INC DISPL POINTER
300         BRA PR2         DO AGAIN
310 PREND   RTS             RETURN TO BASIC
320 ***********************************************
330 *CONVERT HEX DIGITS OF BYTE AT ADDR IN Y TO
340 *THEIR VIDEO CODES AND PUT THEM IN DISPLAY
350 *BUFFER ADDR IN U REG.
360 ***********************************************
380 PR3     LDB ,Y          GET BYTE
390         LDA #$10        MULTIPLIER
400         MUL             SHIFT UPER NIBBLE INTO A
410         CMPA #$09       GREATER THAN 9?
420         BHI PRCVA       IF SO,CONV TO ALPHA
430         ORA #$70        CONV TO NUMERIC CODE
440         BRA PRCVB       GO STORE IT
450 PRCVA   ADDA #$37       CONV TO ALPHA
460 PRCVB   STA PRBYDS,PCR  STORE IT
470         LDA ,Y          GET BYTE
480         ANDA #$0F       GET LOWER NIBBLE
490         CMPA #$09       GREATER THAN 9?
500         BHI PRCVC       IF SO,CNV TO ALPHA
510         ORA #$70        CNV TO NUM CODE
520         BRA PRCVD       GO STORE IT
530 PRCVC   ADDA #$37       CNV TO ALPHA CODE
540 PRCVD   STA 1+PRBYDS,PCR STORE IT
550 ***********************************************
560 *PUT 2 VIDEO CODES AT PRBYDS INTO DISPL BUFFER
570 ***********************************************
580         LDD PRBYDS,PCR  GET VIDEO CODES
590         STD ,U++        PUT INTO BUFFER
600         BRA PR2A        GO TO CONTROLLER
```

```
610 ************************************************
620 PRBYDS   RMB 2          WORKING AREA
630 PRCNT    RMB 1          COUNTER
640 ************************************************
650          END
```

Listing 8-2 The PRFP Subroutine.

This program is relocatable, since the PC relative addressing mode is used. The operand suffix (,PCR) specifies PC relative addressing. All operand addresses of operands inside this program are referenced with this addressing mode. Thus, this program can be loaded at any address that does not conflict with the BASIC program and still work.

At line 200 two bytes are reserved that will receive a pointer from a BASIC program. These two bytes are at locations 2A00 and 2A01 when the subroutine is loaded at address $2A00. This source code should be assembled on tape with the command: A PRFPO/WE. The object code file name is PRFPO. Note that there is no ORG statement in this subroutine.

When the BASIC program and assembled subroutine are run, the binary floating point representation of a number is displayed in hexadecimal. The first byte, reading left to right, is the exponent and the other four bytes are the mantissa. See Chapter 2 for a refresher on binary floating point representation.

**Calling Subroutines (DEFUSRn and USRn)**

Normally a subroutine is called from a BASIC program with the DEFUSRn and USRn functions. The starting address of the subroutine must first be defined by the DEFUSRn function. The format of DEFUSRn is:

DEFUSRn= expression

n indicates one of ten (0 - 9) subroutines. The value of the expression is taken as the starting address of subroutine n. The expression can be a numeric constant, numeric variable, or an arithmetic expression. Examples are:

```
10 DEFUSR0=&H2A00
20 DEFUSR1=K
30 DEFUSR2=&H1000+2*K
```

The above statements define the starting address of three subroutines numbered 0, 1, and 2. Subroutine 0 is specified at line 10 as starting at address $2A00. Subroutine 1 has a starting address equal to the value of K. Subroutine 2 starts at an address equal to $1000 plus two times the value of K.

After the starting address of a particular subroutine is defined, it is called with the USRn function. The n identifier must be the same as the n of the DEFUSRn function. The format of the USRn function in a statement is:

BV = USRn(pass arg)

The (pass arg) is the argument passed to the subroutine; the BASIC variable (BV) is assigned the value returned by the subroutine. Performing a USRn function causes the PC register to be pushed into the S stack and the MPU to be directed to the address specified by DEFUSRn. The argument is passed to the subroutine in the A and X registers. The BASIC program in Listing 8-3 demonstrates calling a subroutine.

```
10 DEFUSR3=&H3000
20 Y=88: X=0
30 X=USR3(Y)
40 PRINT X
```

Listing 8-3 The Use of the DEFUSR and USR Functions.

At line 10 the address of subroutine 3 is defined. At line 20 the value of argument Y is established. The subroutine is called at line 30, where Y is passed to the subroutine. When the subroutine returns to BASIC, statement 40 is performed with the variable X equal to the result of the subroutine. However, it should be understood that not all subroutines require an argument and not all return a result.

The arguments that can be passed to a subroutine with the USRn function are numeric constants and variables, string constants and variables, array elements, and expressions. Some examples are:

```
USR0(125.2)        USR3(3+N/256)
USR1(N$)            USR2("ABCD")
```

The results returned by a subroutine can be a numeric variable, string variable, or an array element. The allowed combinations of pass arguments and receive variable types are:

```
X=USR0(Y)
N=USR1(X$)
M$=USR2(X$)
```

The subroutine has the responsibility of returning a result that matches the receiving variable type. For instance, in the second example above the subroutine receives a string and must return a numeric value.

■ Passing a Numeric Argument to a Subroutine

When control is given to the subroutine, the A and X registers' contents describe the data passed to it. If the passed argument is numeric, the A register will be clear and the X register will contain the pointer to the value. The numeric value will be in a memory area known as the **floating point accumulator (FAC)**. The value is represented in a slightly different form of binary floating point composed of six bytes. This form, as shown in Fig. 8-4, has the exponent in the most significant byte.

$$\underbrace{\text{xxxxxxxx}}_{\text{exponent}}\ \underbrace{\text{xxxxxxxx}\ \ \text{xxxxxxxx}\ \ \text{xxxxxxxx}\ \ \text{xxxxxxxx}\ \ \overset{\text{mantissa sign bit(1=neg; 0=pos)}}{\overset{\searrow}{\text{x}}}\text{xxxxxxx}}_{\text{normalized mantissa}}$$

Fig. 8-4 Binary Floating Point Format in FAC.

The exponent is represented in a type of signed binary where the exponent value is found by subtracting \$80 from the hexadecimal value in the byte. It can also be found by subtracting decimal 128 from the decimal contents of that byte. For example, if the exponent byte contained \$83 then:

$$\text{exponent} = 83_{16} - 80_{16} = 3_{16}$$
or
$$\text{exponent} = 131_{10} - 128_{10} = 3_{10}$$

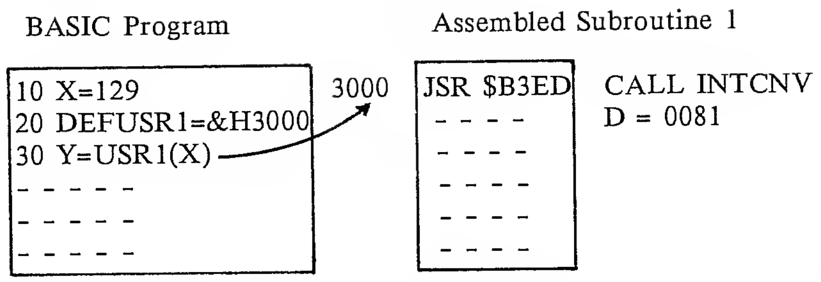
The mantissa is stored in a normalized format in the remaining five bytes. However, the MSB of the last byte is not used to calculate the magnitude of the mantissa. That bit indicates the sign of the mantissa; the sign is positive if clear or negative if set. A value of zero is represented by a 00 exponent byte, and the mantissa is ignored. Here's an example of a number in the FAC:

FAC = 7F C0 00 00 00 80

$$\text{VALUE} = -.11 \times 2^{-1} = -.011_2 = -.375_{10}$$

To summarize: when passing a numeric variable to a subroutine the A register will be clear, indicating a numeric argument has been passed to it, and the X register will point to the FAC which contains the value.

If you don't want to deal with floating point numbers, the INTCNV subroutine can be called by the assembled subroutine. This is a subroutine in BASIC ROM at address \$B3ED that will convert the contents of FAC to a fixed point integer and pass that back to the assembled subroutine in the D register. The value in the D register will be in signed binary, therefore, the range of values is limited to -32768 - +32767 decimal. A demonstration of this can be seen in Fig. 8-5. Observe that the D register, after returning from INTCNV, contains \$0081, the signed binary equivalent of decimal 129.



```
10 X=129
20 DEFUSR1=&H3000
30 Y=USR1(X)
- - - - -
- - - - -
- - - - -
```

```
JSR $B3ED
- - - -
- - - -
- - - -
- - - -
- - - -
```

Fig. 8-5 An Subroutine Calling INTCNV.

The pointer to a numeric variable can be passed to a subroutine. This lets the subroutine work with the variable as it exists in the numeric variable storage area. The BASIC program can pass the pointer to variable X with the statement:

```
50 Y=USR1(VARPTR(X))
```

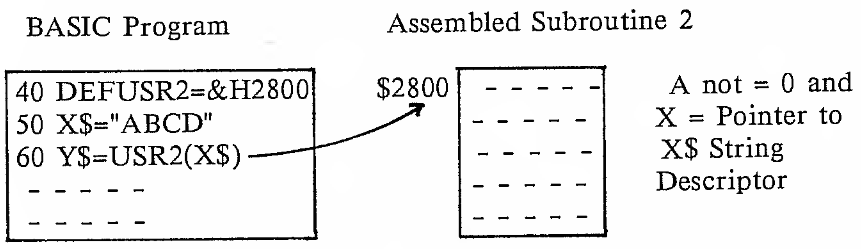
Subroutine 1 must get the pointer by using the INTCNV subroutine:

```
JSR $B3ED
```

This results in the pointer to variable X being in the D register. Now the assembled subroutine can work with variable X as it exists in memory in BASIC's normal five-byte binary floating point format.

■ Passing a String to a Subroutine

When a string is passed to a subroutine, the A register contains a non-zero value to indicate so. The X register will point to the **string descriptor**, which is a five-byte description of the string. The format of the string descriptor is: NN XX AA AA XX. NN is the number of characters in the string and the AAAA field contains the pointer to the string. The XX fields are used by BASIC and should not be changed. A demonstration of passing a string to a subroutine can be seen in Fig. 8-6.



```
40 DEFUSR2=&H2800
50 X$="ABCD"
60 Y$=USR2(X$)
- - - - -
- - - - -
```

Fig. 8-6 Passing a String to a Subroutine.

At line 50 the string X$ is simply defined. In this case that string, ABCD, exists as part of statement 50 in the statement storage area. A string will not be put in the string storage area at the top of available RAM until it has been manipulated in some fashion. Therefore, subroutine 2 in Fig. 8-6 must be very careful when acting upon string X$ so as to not disturb any other statements. If line 50 were changed to: 50 X$="AB"+"CD", the X$ string would reside in the string storage area.

## ■ Returning a Numeric Value to BASIC

A numeric value can be returned to a BASIC program from an assembled subroutine in any of three ways. The three techniques are: returning a value in the FAC, putting a new value in a numeric variable's storage area, and using the GIVABF ROM routine.

The first technique is to put the binary floating point representation of the subroutine's result into the FAC, the area to which the X register points. The data format is as described previously. Then the RTS instruction is executed to return to the BASIC program where the receiving BASIC variable will take on that value. If FAC is not changed, the receiving variable will take on the value of the variable passed to the subroutine when the BASIC program resumes.

The second technique is used when the pointer to a numeric variable is passed to a subroutine. The pointer tells the subroutine where the numeric variable is located in the numeric variable storage area. Then the newly calculated value can be put into memory at that address using BASIC's normal five-byte binary floating point format. Upon returning to the BASIC program via an RTS instruction, the variable whose pointer was passed will be the value generated by the subroutine and the receiving variable will take on the value of the pointer. For example:

```
10 DEFUSR1=&H2800
20 Y=6544: X= -12
30 X=USR1(VARPTR(Y))
40 PRINT X,Y
```

At line 40 X equals the value of the pointer to Y, and Y equals the value that subroutine 1 put into memory.

The third method is for the subroutine to jump to GIVABF, a ROM resident program at address $B4F4. GIVABF takes the contents of the D register in signed binary and returns that integer value to the receiving variable. The range of values is limited to -32768 - +32767 decimal. This return technique can also be used when a string has been passed to a subroutine which is to return a numeric value. An RTS instruction is not needed because the GIVABF program performs that function. This technique is demonstrated in Fig. 8-7. At line 40 the variable K equals decimal 23, the equivalent of $0017 that was put into the D register by subroutine 3.

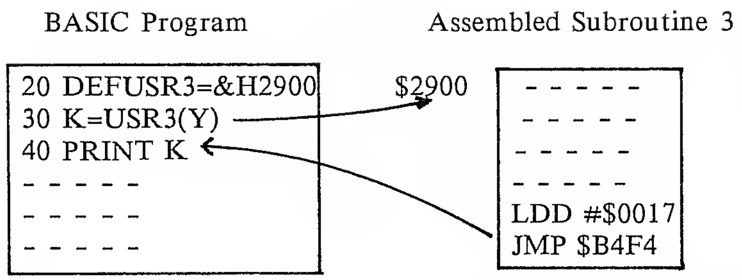


Fig. 8-7 Returning An Integer Value Using GIVABF.

■ Returning a String to BASIC

An subroutine can return a string to be assigned to a BASIC string variable. First, the BASIC program should establish a string area for the return string variable, and that area must be large enough for the largest string the subroutine may return. This can be done with the STRING$ command:

```
R$=STRING$(xxx," ")
```

This statement establishes a string made up of xxx spaces that a subroutine can fill with its return string. The subroutine is then called with the statement: R$=USR4(R$), which passes the null string, R$, to the subroutine. Upon returning, R$ equals whatever the subroutine has placed in that string area. This process can also be done in one statement:

```
R$=USR4(STRING$(xxx," "))
```

The receiving string is set equal to the passed string after it has been modified by the assembled subroutine.

One must take care to **never** lengthen a BASIC string with a subroutine, for that will cause that string to overlay some other string in memory. Remember that the BASIC strings are packed together in the string storage area at the top of available RAM.

Presented in Listing 8-4 is the source code of a subroutine that will fill or modify a BASIC string by using the string descriptor to tell it where the string is in memory. It will then shorten the string length to reflect its new contents by modifying the length code within the string descriptor.

The source code should be assembled with the command: A/IM/WE/AO, resulting in the object code in memory at address $3000. Verify that no errors were detected during assembling. Then go to BASIC by the Q command. Enter the BASIC program in Listing 8-5. This BASIC program calls the assembled subroutine and prints the returned string R$.

```
100 *THIS SUBROUTINE WILL FILL A BLANK STRING WITH
110 *THE WORDS "ANY STRING WILL DO" AND SET THE
120 *LENGTH DESCRIPTOR TO 19, THE NUMBER OF
130 *CHARACTERS IN THE STRING.
140 **********************************************
150          ORG $3000
160 STR      LDY 2,X         GET ADDR OF STRNG
170          LDU #STEXT      GET TEXT ADDR
180          LDA #19         COUNT VALUE
190 ST1      LDB ,U+         GET NEXT TEXT CHAR
200          STB ,Y+         PUT IN STRING
210          DECA            DEC COUNT
220          BNE ST1         DO AGAN IF NOT DONE
230          LDA #19         STRNG LENGTH
240          STA ,X          PUT IN DESCRIPTOR
250          RTS             RETURN TO BASIC
260 **********************************************
270 STEXT    FCC /ANY STRING WILL DO/
280          END
```

Listing 8-4 A Subroutine to Fill a String.

```
 5 CLEAR 200,&H2F00
10 DEFUSR0=&H3000
20 R$=USR0(STRING$(50," "))
30 PRINT R$
40 END
```

Listing 8-5 A BASIC Program Calling a Subroutine.

The technique of assembling programs directly into memory not used by BASIC is quite helpful when testing and debugging programs and subroutines. In this fashion one can quickly and easily put a program into memory for immediate testing. However, when going from EDTASM+ to BASIC the source code will be lost, and on the reverse path the BASIC program will be lost.

## SUBROUTINES IN BASIC ROM

The Extended Color BASIC system in ROM is composed of many subroutines. They can be used by assembly language programs if the programmer knows their operating details such as their starting address, how to pass parameters to them, and what functions they perform. Presented in this section are a limited number of ROM subroutines that deal with **input** and **output**, or I/O. Inputting is the transferring data from a device external to the MPU and memory, such as the keyboard, or tape recorder, into memory or an MPU register. Outputting is transferring data from memory or an MPU register to an external device. There are many more subroutines in BASIC ROM that serve useful purposes, such as generating graphics displays, converting binary floating point to decimal, and vice versa. You can use ZBUG to look through ROM to try to locate them and also to find out how to use them.

The ROM subroutines are convenient since they already exist and do not use much RAM because they reside in ROM. However, the subroutines do not save the MPU registers they use. This can be remedied by constructing a subroutine that stores the required registers, calls the ROM subroutine, and then restores the registers before returning to the calling program. This technique will be demonstrated for each of the ROM subroutines presented if necessary. The subroutines are presented in related groups. Those that deal with cassette I/O are presented in the Cassette section, for example.

**Display and Print Subroutines**

Three ROM subroutines are presented here: CLSCRN, CHROUT, and DISPL. They all control the text screen. The CHROUT subroutine can also output data through the **serial** I/O, or **RS-232**, port, to which a printer is normally connected but other devices, such as a modem which accepts serial data, can be connected to it. Serial data transmission is a technique of transferring binary data from one electronic device to another. A byte is transmitted serially by sequentially sending out on a wire the state of each bit of that byte. First, the LSB of a byte is sent out, then the more significant bits are sent in the following order; 0, 1, 2, 3, 4, 5, 6, and 7 last. This process repeats for the next byte to be sent out.

Computers with Color BASIC 1.0 send out only seven bits of each byte, bits 0 - 6, so only the 128 ASCII codes may be transmitted. Color BASIC 1.1 and beyond send out all eight bits, so any value in a byte may be transmitted. This can be determined for your computer by entering the BASIC command: EXEC 41175. This causes the Color BASIC descriptive header to be displayed, and you can see which version you have. The rate at which bits are transmitted is called **baud**. A baud of 600 bits per second will transmit about 75 bytes per second.

■ CLSCRN

Function: Clear the text screen and set the display pointer to the first display position.

Address: $A928 S Stack Use: None

Registers Modified: B, X, and CC

Pass Arguments: None

Return Values: Display pointer set to $0400.

The CLSCRN subroutine clears the text screen and sets the display pointer to $0400. The **display pointer** is the content of addresses $88 and $89, which points to the display position on the screen at which the next character will be displayed. The value of the display pointer can range from $0400 - $05FF, the addresses of the text display buffer. BASIC's CLS command is similar to CLSCRN; the screen is cleared and the next text displayed with the PRINT command starts at the top left corner of the screen.

The CLSCRN subroutine does not preserve the B, X, and CC registers. Listing 8-6 is a subroutine to use with an assembly language program that preserves the registers and calls CLSCRN.

```
5000 *SUBROUTINE NAME: CLS
5010 *THIS SUBROUTINE WILL CLEAR THE TEXT SCREEN
5020 *AND RESET THE DISPLAY POINTER.  6 BYTES OF
5030 *THE S STACK ARE USED.
5040 *********************************************
5050 CLS     PSHS B,X,CC     SAVE REGS
5060         JSR $A928       CALL CLSCRN
5070         PULS B,X,CC     RESTORE REGS
5080         RTS             RETURN
5090 *********************************************
```

Listing 8-6 The CLS Subroutine.

The CLS subroutine uses six bytes of the S stack; four for storing the B, X, and CC registers, and two for storing PC when CLSCRN is called. This does not include the two bytes used to store PC when this subroutine, CLS, is called.

## ■ CHROUT

Function: CHROUT will output a character, specified by its ASCII code in the A register, to the text screen or out the serial I/O port.

Address: [$A002]  S Stack Use: 8 bytes

Registers Modified: CC

Pass Parameters: The DP register must contain a 00. The contents of the A register is outputted. The contents of address $6F specifies output device.

$6F = 00; output to the screen.

$6F = FE; output through serial I/O port.

Return Values: If $6F = 00, the display pointer is incremented.

The starting address of CHROUT is contained in addresses $A002 and $A003; it can be called by using extended indirect addressing.

JSR [$A002]  or  JSR $A282

One can inspect the contents of $A002 and $A003 with ZBUG in the word display mode to find the absolute starting address. That address is normally $A282, so CHROUT can also be called with extended addressing. This results in a slightly faster execution speed since a jump with extended addressing is faster than with indirect addressing. If one wants the CC register preserved, CHROUT should be called from within a subroutine that will save and then restore the CC register.

If location $6F contains a 00, CHROUT will display a character whose

ASCII code is in the A register on the text screen at the position specified by the display pointer. Each time a character is displayed, the display pointer is incremented by one. When the display pointer is incremented past $5FF, CHROUT will scroll all the text on the screen up one line.

Listing 8-7 is a subroutine that displays a character on the text screen. It would be used in a situation where the calling program generates the characters one at a time. Note that line 5190 can be changed to extended addressing by using the starting address contained within $A002 and $A003.

```
5100 *SUBROUTINE NAME: ACHRO
5110 *THIS SUBROUTINE WILL DISPLAY THE CHARACTER
5120 *WHOSE ASCII CODE IS IN THE A REGISTER AT THE
5130 *POSITION SPECIFIED BY THE DISPLAY POINTER.
5140 *THEN THE DISPLAY POINTER IS INCREMENTED.
5150 *13 BYTES OF STACK AREA ARE REQUIRED.
5160 ***********************************************
5170 ACHRO   PSHS CC,B,DP     SAVE REGS
5180         CLRB             GET 00
5190         TFR B,DP         CLEAR DP REG
5200         CLR $006F        OUTPUT TO SCREEN
5210         JSR [$A002]      CALL CHROUT
5220         PULS CC,B,DP     RESTORE REGS
5230         RTS              RETURN
5240 ***********************************************
```

Listing 8-7 The ACHRO Subroutine.

If location $6F contains $FE, the contents of A will be sent out the serial I/O port at the baud specified by the contents of addresses $95 and $96. The baud can be set to the desired rate by putting the selected value from Table 8-1 into locations $95 and $96 before calling CHROUT. BASIC and EDTASM+ initially set the baud to 600 bits per second. It is the programmer's responsibility to set the baud to match the rate at which the device accepts data.

| Baud | Hex. Value |
|---|---|
| 120 | 01CA |
| 300 | 00BE |
| 600 | 0057 |
| 1200 | 0029 |
| 2400 | 0012 |
| 4800 | 0006 |
| 9600 | 0001 |

Table 8-1 Baud Control Values.

If a printer is connected to the serial I/O port, it will print the character whose ASCII code is sent out. Other devices will interpret the byte sent according to their design.

Listing 8-8 is a subroutine that preserves the CC register and calls CHROUT to output a byte on the serial I/O port.

```
5300 *SUBROUTINE NAME: BCHRO
5310 *THIS SUBROUTINE WILL OUTPUT THE CONTENTS
5320 *OF A TO A DEVICE ON THE SERIAL PORT. THE
5330 *BAUD RATE SHOULD HAVE BEEN PREVIOUSLY
5340 *DETERMINED. 13 BYTES OF STACK ARE USED.
5350 ******************************************
5360 BCHRO   PSHS CC,B,DP    SAVE REGS
5370         CLRB            GET 00
5380         TFR B,DP        CLEAR DP REG
5390         LDB #$FE        SERIAL OUTPUT
5400         STB $006F       PASS PARAM
5410         JSR [$A002]     CALL CHROUT
5420         PULS CC,B,DP    RESTORE REGS
5430         RTS             RETURN
5440 ******************************************
```

Listing 8-8 The BCHRO Subroutine.

■ DISPL

Function: DISPL displays a string of characters on the text screen, starting at the position specified by the display pointer.

Address: $B99C S Stack Use: 10 bytes

Registers Modified: A, B, X, U, and CC.

Pass Parameters: The DP register must contain 00. The X register points to the string to be displayed. The display pointer specifies starting display position.

Return Values: The display pointer is incremented.

The DISPL subroutine displays a string of characters on the text screen, starting at the position specified by the display pointer. The string to be displayed must exist in memory as a string of ASCII codes. The X register must be loaded with the starting address minus one, of the string to be displayed. DISPL does not display the first character that X points to, but just those thereafter. This is because this subroutine is used by BASIC to print strings; each string exists between a pair of quotation marks. When a string is printed by BASIC, the opening quotation mark is not printed. The characters that can be displayed are all the displayable text characters plus the graphics characters, except the quotation mark. The end of a string is identified by a quotation mark: an ASCII code of $22. Upon encountering a quotation mark, DISPL stops displaying characters and returns to the calling program. Thus, quotation marks can not be displayed. A byte in the string, of the value $0D, causes DISPL to perform a carriage return and a line feed (CR/LF). The next display position will be at the beginning of the next line. As DISPL is displaying a string, the display pointer is incremented by one for each character displayed. After performing a carriage return and line feed, the display pointer points to the first position of the next line. If the display pointer is incremented beyond a value of $5FF, the text on the screen is scrolled up one line to make room for the next line of text.

Listing 8-9 is a subroutine that can be used with assembly language programs, since it preserves all the registers. It will decrement X by one before calling DISPL. Thus, X can point directly to the first character of the string when calling the DSPLAY subroutine.

```
5500 *SUBROUTINE NAME: DSPLAY
5510 *THIS SUBROUTINE WILL DISPLAY THE TEXT
5520 *STRING THAT THE X REGISTER POINTS TO
5530 *STARTING AT THE POSITION SPECIFIED BY
5540 *THE DISPLAY POINTER. A CODE OF 0D WILL
5550 *CAUSE A CR/LF. A CODE OF 22 (") INDICATES
5560 *THE END OF THE STRING TO DISPLAY.  20
5570 *BYTES OF STACK AREA ARE REQUIRED.
5580 *******************************************
5590 DSPLAY  PSHS D,X,U,CC,DP SAVE REGS
5600         CLRB            GET 00
5610         TFR B,DP        CLEAR DP REG
5620         LEAX -1,X       ADJUST POINTER
5630         JSR $B99C       CALL DISPL
5640         PULS D,X,U,CC,DP RESTORE REGS
5650         RTS             RETURN
5660 *******************************************
```

Listing 8-9 The DSPLAY Subroutine.

**Reading The Keyboard**

Interrogating the keyboard to determine if or what key has been depressed is often done by programs. Fortunately, there is a subroutine in BASIC ROM that performs this function. It is called POLCAT.

■ POLCAT

Function: Interrogate the keyboard to see what key is being depressed, if any.

Address: [$A000] S Stack Use: 12 bytes

Registers Modified: A and CC.

Pass Parameters: None.

Return Values: If a key is depressed Z bit = 0, and A register contains its ASCII code. If no key is depressed Z bit = 1, and the A register contains 00.

The POLCAT subroutine samples the keyboard and returns the ASCII code of a character in A if a key was depressed. It also indicates whether a key was depressed with the Z bit of the CC register. The starting address of POLCAT is contained in locations $A000 and $A001. Thus, POLCAT can be called using extended indirect addressing.

JSR [$A000]    or    JSR $A1C1

Or the absolute address ($A1C1 in Color BASIC 1.0 or 1.1) of POLCAT can be found and extended addressing used.

The A and CC registers are modified by this subroutine. Since the return values are returned in A and CC, they do not have to be saved and restored. Therefore, a subroutine does not have to be constructed to save and restore any registers. POLCAT will require 12 bytes of stack area for its own use plus two more in which to store PC when it is called.

Listing 8-10 uses the POLCAT, CLS, ACHRO, and DSPLAY subroutines. It clears the screen and displays the string: ENTER A DECIMAL NUMBER. Then it polls the keyboard and accepts only the digits 0 - 9, which it displays. When the ENTER key is hit, it causes the program to return to ZBUG.

```
 100 *PROGRAM NAME: ADEMO
 110 *THIS PROGRAM WILL DEMONSTRATE THE USE OF THE
 120 *CLS, ACHRO, DSPLAY, AND POLCAT SUBROUTINES.
 130 *IT WILL ESTABLISH ITS OWN STACK OF 22 BYTES.
 140 ***********************************************
 150          ORG $2500
 160 ADEMO    STS ADOS        SAVE ZBUG S REG
 170          LDS #22+ADST    ESTABLISH STACK
 180          JSR CLS         CLEAR SCREEN
 190          LDX #ADTEXT     GET TEXT POINTER
 200          JSR DSPLAY      DISPLAY TEXT STRING
 210 AD1      JSR [$A000]     POLL KEYBOARD
 220          BEQ AD1         DO AGAIN IF NO KEY
 230          CMPA #$0D       ENTER KEY?
 240          BEQ AD2         IF SO,GOTO END
 250          CMPA #$30       LESS THAN 0 KEY
 260          BLO AD1         GET NEW KEY
 270          CMPA #$39       GREATER THAN 9 KEY
 280          BHI AD1         GET NEW KEY
 290          JSR ACHRO       DISPLAY CHARACTER
 300          BRA AD1         GET NEXT CHARACTER
 310 AD2      LDS ADOS        RESTORE ZBUG S REG
 320          SWI             RETURN TO ZBUG
 330 ***********************************************
 340 ADOS     RMB 2
 350 ADST     RMB 22
 360 ADTEXT   FCC /ENTER A DECIMAL NUMBER "/
 370 ***********************************************
5000 *SUBROUTINE NAME: CLS
5010 *THIS SUBROUTINE WILL CLEAR THE TEXT SCREEN
5020 *AND RESET THE DISPLAY POINTER.  6 BYTES OF
5030 *THE S STACK ARE USED.
5040 ***********************************************
5050 CLS      PSHS B,X,CC     SAVE REGS
5060          JSR $A928       CALL CLSCRN
5070          PULS B,X,CC     RESTORE REGS
5080          RTS             RETURN
5090 ***********************************************
5100 *SUBROUTINE NAME: ACHRO
```

```
5110 *THIS SUBROUTINE WILL DISPLAY THE CHARACTER
5120 *WHOSE ASCII CODE IS IN THE A REGISTER AT THE
5130 *POSITION SPECIFIED BY THE DISPLAY POINTER.
5140 *THEN THE DISPLAY POINTER IN INCREMENTED.
5150 *13 BYTES OF STACK AREA ARE REQUIRED.
5160 ***********************************************
5170 ACHRO   PSHS CC,B,DP    SAVE REGS
5180         CLRB            GET 00
5190         TFR B,DP        CLEAR DP REG
5200         CLR $006F       OUTPUT TO SCREEN
5210         JSR [$A002]     CALL CHROUT
5220         PULS CC,B,DP    RESTORE REGS
5230         RTS             RETURN
5240 ***********************************************
5500 *SUBROUTINE NAME: DSPLAY
5510 *THIS SUBROUTINE WILL DISPLAY THE TEXT
5520 *STRING THAT THE X REGISTER POINTS TO
5530 *STARTING AT THE POSITION SPECIFIED BY
5540 *THE DISPLAY POINTER. A CODE OF 0D WILL
5550 *CAUSE A CR/LF. A CODE OF 22 (") INDICATES
5560 *THE END OF THE STRING TO DISPLAY.  20
5570 *BYTES OF STACK AREA ARE REQUIRED.
5580 ***********************************************
5590 DSPLAY  PSHS D,X,U,CC,DP SAVE REGS
5600         CLRB            GET 00
5610         TFR B,DP        CLEAR DP REG
5620         LEAX -1,X       ADJUST POINTER
5630         JSR $B99C       CALL DISPL
5640         PULS D,X,U,CC,DP RESTORE REGS
5650         RTS             RETURN
5660 ***********************************************
5670         END
```

Listing 8-10 The ADEMO Program.

The stack area is set to 22 bytes; 20 for DSPLAY plus 2 for storing PC when DSPLAY is called. Assemble the source code into memory with the command: A/IM/WE/AO and verify there are no errors. Then go to ZBUG and run the program by entering: GADEMO. Notice that only decimal numbers are accepted from the keyboard and displayed.

**Reading The Joysticks**

Two joysticks can be connected to the Color Computer and their positions read with the JOYIN ROM subroutine. It is also possible to connect other devices to the joystick ports as long as they are electronically compatible with the Color Computer. JOYIN can then be used to read the values originating at those devices.

There is no ROM subroutine to read the state of a joystick fire button. The state of a joystick fire button can be read by reading the contents of address $FF00, a **dedicated** address. A dedicated address is neither RAM nor ROM, but is connected to some electronic device. The fire buttons are connected to bits 0 and 1 of address $FF00. Bit 0 reflects the state of the right fire button and bit 1 reflects the state of the left fire button. If the fire button is depressed, the bit is clear; otherwise it is set.

■ JOYIN

Function: This subroutine reads the position of both joysticks and returns their positions as numeric values in specific memory locations.

Address: [$A00A] S Stack Use: 6 bytes

Registers Modified: A,B,X,U,CC.

Pass Parameters: None.

Return Values: In four memory locations, $15A through $15D.

|  |  |
|---|---|
| Right joystick: | $015A = left/right position value. |
|  | $015B = up/down position value. |
| Left joystick: | $015C = left/right position value. |
|  | $015D = up/down position value. |

A value of 00 indicates all the way up or left, and a value of $3F indicates all the way down or right.

The starting address of JOYIN is contained in addresses $A00A and $A00B. Therefore, it can be called using extended indirect addressing.

```
JSR  [$A00A]     or     JSR  $A9DE
```

If the contents of $A00A and $A00B are displayed, the absolute address (typically $A9DE) can be found and extended addressing can be used when calling it.

Upon returning from JOYIN, values indicating the joystick positions will be in memory. For example, if the left joystick was centered, then $15C and $15D would each contain $1F (a value half-way between 00 and $3F). If the right joystick was positioned to the extreme upper right, location $15A would contain $3F and $15B would contain 00.

The JOYIN subroutine modifies the A, B, X, U, and CC registers, thus, a subroutine should be constructed that saves these registers, calls JOYIN, and then restores the same registers. Such a subroutine (JOY) is in Listing 8-11.

Listing 8-11 is a program that demonstrates using the JOY subroutine. It will display a cursor on a clear screen that moves as the left joystick is moved throughout its range. When the joystick is centered, the cursor is displayed in the center of the screen. This program can be used to test the joysticks for proper operation; however, the fire button is not tested. The JDEMO program can be made to use the right joystick by changing lines 220 and 270 to the following:

```
220          LDB $015B
270          LDB $015A
```

Assemble the JDEMO program with the A/IM/WE/AO command and verify there are no errors. Then go to ZBUG and run it by entering: GJDEMO. The cursor on the screen will now move as the joystick is moved.

```
 100 *PROGRAM NAME: JDEMO
 110 *THIS PROGRAM WILL POSITION A CURSOR ON THE
 120 *SCREEN TO MATCH THE PHYSICAL POSITION OF
 130 *THE LEFT JOYSTICK.
 140 ********************************************
 150         ORG $2500
 160 JDEMO   LDS #17+JDSTCK  ESTABLISH STACK
 170         JSR $A928       CLEAR THE SCREEN
 180         LDX #$04E0      CENTER OF SCREEN
 190 JD1     TFR X,U         STORE POINTER
 200         LDY #$0400      START OF SCREEN
 210         JSR JOY         GET JSTICK VALUES
 220         LDB $015D       GET VERTICAL POSITION
 230         LDA #$08        SHIFT VALUE
 240         MUL             SHIFT
 250         ANDB #$E0       CALC VERT OFFSET
 260         LEAY D,Y        STORE VERT OFFSET
 270         LDB $015C       GET HORIZ POSITION
 280         LDA #$80        SHIFT VALUE
 290         MUL             SHIFT
 300         TFR A,B         PUT IN LOWER D REG
 310         CLRA            CLEAR UPPER D
 320         LEAY D,Y        STORE HORIZ OFFSET
 330         TFR Y,X         SAVE NEW POSITION
 340         LDA #$60        SPACE CODE
 350         STA ,U          DISPLAY IT
 360         LDA #128        CURSOR CODE
 370         STA ,X          DISPLAY IT
 380         BRA JD1         DO AGAIN
 390 ********************************************
 400 JDSTCK  RMB 17          STACK AREA
 410 ********************************************
5700 *SUBROUTINE NAME: JOY
5710 *THIS SUBROUTINE WILL RETURN THE LEFT AND
5720 *RIGHT JOYSTICK POSITIONS. IT CALLS JOYIN
5730 *IN ROM. 15 BYTES OF STACK AREA ARE REQUIRED.
5740 ********************************************
5750 JOY     PSHS D,X,U,CC   SAVE REGS
5760         JSR  [$A00A]    CALL JOYIN
5770         PULS D,X,U,CC   RESTORE REGS
5780         RTS             RETURN
5790 ********************************************
5800         END
```

Listing 8-11 The JDEMO Program.

## Cassette Tape I/O

Four subroutines in BASIC ROM are used to write and read data to and from tape. They are WRTLDR (write leader), BLKOUT (write block out), CSRDON (start to read), and BLKIN (read block in). First let's make sure you understand the conventions established for tape I/O.

Data is recorded on tape in a series of blocks or groups of bytes along the length of the tape. Extended Color BASIC and EDTASM+ use common conventions to determine the format of the blocks as they are recorded on tape. The same conventions can be used to read data from tape. A group of blocks on tape comprises a **file**, and all the blocks contain the information in that file. A typical tape file is shown in Fig. 8-8.

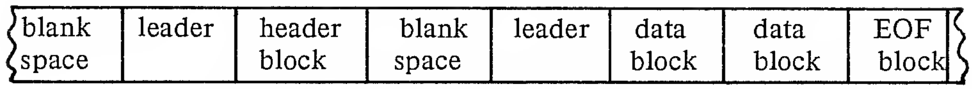


Fig. 8-8 A BASIC Cassette Tape File.

The file is written and read from left to right; the leader is encountered first. The leader is used when reading the tape. It synchronizes the cassette tape read electronics and the subroutine to read the following block. The leader is a series of 128 bytes whose value is $55. Immediately following the leader is the header, or name, block which describes the file. Within the header is information such as the file name; whether it is a BASIC program, data, or object code file; and where to load into memory. Following the header is a space in which nothing is recorded. This blank area serves as a time delay of about one-half second, giving BASIC time to evaluate the header block before the data blocks arrive.

Before the data blocks are read there is a another leader to get the read logic in sync. The data blocks contain file data, such as a BASIC program, object code, or data for a table or array. The last block is the end-of-file (EOF) block.

Each type of block has a specific internal structure. The three types are the header, data, and EOF blocks. The following descriptions of the blocks give the particular conventions used by BASIC and EDTASM+. One may choose to use different formatting conventions when writing tape files with assembly language programs, but only you and whoever knows your conventions will be able to read them.

The header block is the most complicated because it is composed of the largest number of fields. The header block format can be seen in Fig. 8-9.

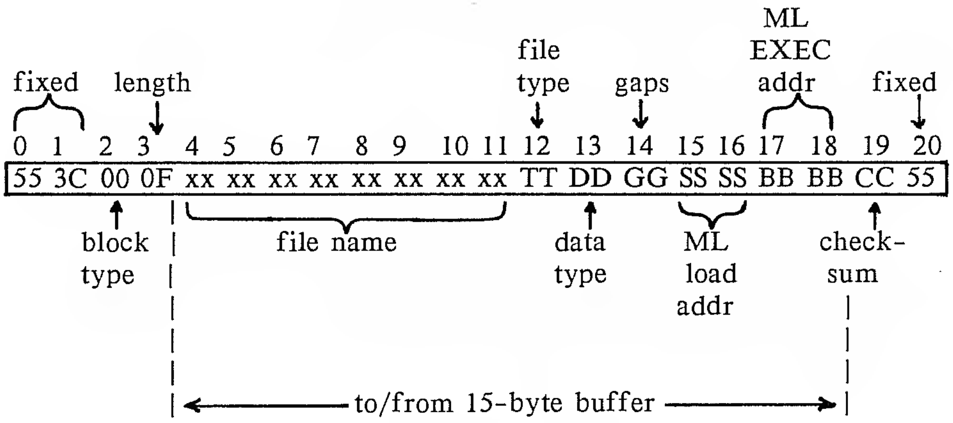
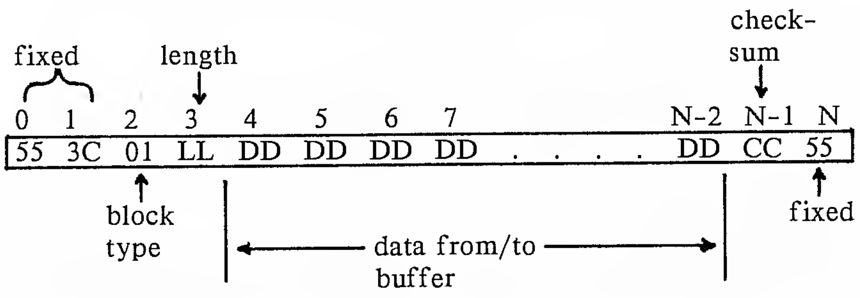


Fig. 8-9 Header Block Format.

The bytes of a header block are described as follows:

- 0,1 - Always written as $55 $3C by subroutine that creates a block.
- 2 - Block type. 00 indicates a header block.
- 3 - Length code. Always set to 15 ($0F) for a header block.
- 4-18 - Data bytes. This data comes from a 15-byte buffer when the header block is written. When read, data goes into a 15-byte buffer.
  - 4-11 - The file name.
    - 12 - File type: 00=BASIC program file, 01=data file, 02=object code file.
    - 13 - Data type: 00=binary. FF=ASCII (as with a CSAVE "xxx",A).
    - 14 - Gaps between data blocks indicator. 00=no gaps (as in Fig. 8-8). FF=yes gaps (as in Fig. 8-12).
  - 15,16 - Address at which to load an object code file.
  - 17,18 - Address at which to start executing object code.
- 19 - Checksum byte. It contains the binary sum of the contents of bytes 2 - 18, with all carries from bit 7 ignored. It is used to verify that data read in is correct.
- 20 - Always $55.



fixed
length
check-sum
0 1 2 3 4 5 6 7 N-2 N-1 N
55 3C 01 LL DD DD DD DD . . . . . DD CC 55
block type
data from/to buffer
fixed

Fig. 8-10 Data Block Format.

The bytes in a data block, shown in Fig. 8-10, are described as follows:

- 0,1 - Always written as $55 $3C by subroutine that creates a block.
- 2 - Block type: 01 = data block.
- 3 - Number of bytes of data in this block (00 - FF).
- 4-(N-2) - Data in the data block. This data comes from a buffer when the block is written. Upon reading, the data is read into a buffer area.
- N-1 - Checksum byte containing the binary sum of bytes 2 through (N-2). Used to verify that the block was read correctly.
- N - Always $55.

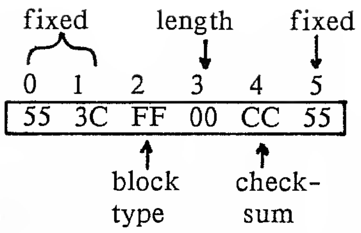


Fig. 8-11 EOF Block Format.

The bytes of an EOF block, shown in Fig. 8-11, are described below. Notice that an EOF block contains no data bytes.

0,1 - Always written as $55 $3C by subroutine that creates a block.
 2 - Block type; FF indicates an EOF block.
 3 - Length code. Must be 00 for an EOF block.
 4 - Checksum byte containing the binary sum of bytes 2 and 3. Used to verify that this block was read correctly.
 5 - Always $55.

The two ROM subroutines WRTLDR and BLKOUT are used to write a tape file. WRTLDR turns on the cassette motor and writes a blank space followed by a leader. Then BLKOUT should be immediately used to write the following block. CSRDON and BLKIN read a tape file. CSRDON turns on the cassette motor and gets the read circuits and subroutine in sync while passing over the leader. BLKIN is used when the cassette is at speed and in sync to read a header, data, or EOF block.

It is up to the operator to place the cassette recorder in its proper condition before reading or writing. Remember that a cassette recorder erases old data when it records new data. Leave 5 to 10 seconds of blank playing time between files; this will make it much easier to position the tape to read a file.

Turning the cassette motor off is done with a dedicated address. Bit 3 of address $FF21 controls the cassette motor. Setting that bit turns the motor on and clearing it turns the motor off. Be careful not to affect any other bits since they control other operations in the computer. Bit 3 can be cleared with the following three instructions:

```
LDA $FF21
ANDA #$F7
STA $FF21
```

without disturbing the other bits in $FF21.

The cassette motor needs to be turned off when the EOF block has been written, or when a data or header block has been written and it will be some time before the next data block will be written. This would happen in a program that generates data to be recorded at a slow rate. After stopping the tape and accumulating a full data buffer, the next block is written by writing a leader and then the data block. This is done because the cassette read logic goes out of sync when the tape stops. A file created by this process contains gaps between the data blocks as seen in Fig. 8-12.

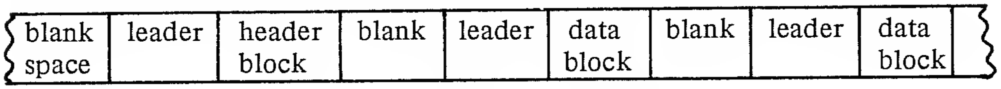


Fig. 8-12 A Tape File with Gaps.

Tape files created with BASIC's CSAVE "xxx",A command and EDTASM+'s assemble onto tape (A) and write source code to tape (W) commands are written with gaps. Tape files created with BASIC's CSAVE "xxx" and CSAVEM commands and EDTASM+'s write memory to tape (P) command are written without gaps.

■ WRTLDR
Function: Turn on cassette motor and write a gap and leader.
Address: [$A00C] S Stack Use: 4 bytes
Registers Modified: A,B,X,Y,CC.
Pass Parameters: The DP register must contain 00.
Return Values: None.

WRTLDR turns on the cassette motor, writes a blank space, and writes a leader. When it returns the program must be ready to write a block.

Since WRTLDR preserves only the U and S registers, it should be called from a subroutine that saves the other registers. One should ensure the DP register is clear before calling WRTLDR or it will not work (see Listing 8-12).

```
5800 *SUBROUTINE NAME: WLDR
5810 *THIS WILL TURN ON THE CASSETTE MOTOR AND
5820 *WRITE A GAP AND LEADER. 14 BYTES OF STACK
5830 *AREA ARE REQUIRED.
5840 ******************************************
5850 WLDR    PSHS D,X,Y,DP,CC SAVE REGS
5860         CLRB             GET 00
5870         TFR B,DP         CLEAR DP REG
5880         JSR  [$A00C]     CALL WRTLDR
5890         PULS D,X,Y,DP,CC RESTORE REGS
5900         RTS              RETURN
5910 ******************************************
```

Listing 8-12 The WLDR Subroutine.

■ BLKOUT

Function: It writes a header, data, or EOF block on tape, specified by the pass parameters.

Address: [$A008] S Stack Use: 10 bytes

Registers Modified: A,B,X,Y,CC.

Pass Parameters: The cassette must be at speed and have completed writing a leader or block. The DP register must contain 00. The controlling parameters are passed in the following memory locations:

$7C = block type to write; 00 = header, 01 = data, FF = EOF;
$7D = block buffer length; range from 00 to $FF;
$7E+$7F= starting address of buffer.

Return Values: The X register contains the sum of the buffer starting address and the buffer length.

Before calling BLKOUT, the parameters should be put in addresses $7C through $7F to tell it what type of block to write. The data to be written should be in the buffer area. After writing an EOF block, the cassette motor should be turned off. If it will be some time between data blocks before there is more data to write, the motor should also be turned off. In this case, another leader must be written before the next data block is written.

Presented in Listing 8-13 are two subroutines, WBLOK and MOTOFF. WBLOK writes a block by calling BLKOUT. MOTOFF turns the cassette motor off.

```
6000 *SUBROUTINE NAME: WBLOK
6010 *THIS WILL WRITE A HEADER, DATA, OR EOF BLOCK
6020 *ON CASSETTE TAPE. ADDRESSES 7C=BLOCK TYPE;
6030 *7D=BUFFER LENGTH; 7E+7F=BUFFER ADDRESS.
6040 *20 BYTES OF STACK AREA ARE REQUIRED.
6050 ********************************************
6060 WBLOK   PSHS D,X,Y,DP,CC SAVE REGS
6070         CLRB             GET 00
6080         TFR B,DP         CLEAR DP REG
6090         JSR  [$A008]     CALL BLKOUT
6100         PULS D,X,Y,DP,CC RESTORE REGS
6110         RTS              RETURN
6120 ********************************************
6200 *SUBROUTINE NAME: MOTOFF
6210 *THIS SUBROUTINE WILL TURN OFF THE CASSETTE
6220 *MOTOR. TWO BYTES OF STACK AREA ARE USED.
6230 ********************************************
6240 MOTOFF  PSHS A,CC        SAVE REGS
6250         LDA $FF21        GET CONTROL BYTE
6260         ANDA #$F7        CLEAR BIT 3
6270         STA $FF21        STORE CONTRL BYTE
6280         PULS A,CC        RESTORE REGS
6290         RTS              RETURN
6300 ********************************************
```

Listing 8-13 The WBLOK and MOTOFF Subroutines.

■ CSRDON

Function: Turns on cassette motor and gets cassette in sync for reading by reading the leader.

Address: [$A004] S Stack Use: 6 bytes

Registers Modified: A,B,X,CC.

Pass Parameters: The DP register must contain 00.

Return Values: None.

CSRDON turns on the cassette motor and searches for a leader. Reading the leader gets the read electronics and program in sync to read the block that follows. CSRDON should be used when starting to read a file and when about to read a data block, if the cassette motor is off. Listing 8-14 is a subroutine that preserves the registers and calls CSRDON.

```
6400 *SUBROUTINE NAME: RLDR
6410 *THIS WILL TURN ON THE CASSETTE MOTOR AND
6420 *USE THE LEADER TO GET IN SYNC FOR READING
6430 *THE FOLLOWING BLOCK. 14 BYTES OF STACK
6440 *AREA ARE USED.
6450 *****************************************
6460 RLDR    PSHS D,X,CC,DP  SAVE REGS
6470         CLRB            GET 00
6480         TFR B,DP        CLEAR DP REG
6490         JSR  [$A004]    CALL CSRDON
6500         PULS D,X,CC,DP  RESTORE REGS
6510         RTS             RETURN
6520 *****************************************
```

Listing 8-14 The RLDR Subroutine.

■ BLKIN

Function: Reads a header, data, or EOF block from tape if the motor is on and the cassette in sync.

Address: [$A006] S Stack Use: 8 bytes

Registers Modified: A,B,X,CC.

Pass Parameters: The DP register must contain 00. The buffer starting address must be put in addresses $7E and $7F.

Return Values: The data is read from tape into the buffer. The block type and length of the block read is returned in memory addresses $7C and $7D:

$7C = Block type: 00 = Header block, 01 = Data block, FF = EOF block.

$7D = Block length.

The completion status is returned in the CC and A registers:

Z=1 and A=00: indicate a successful read.

Z=0 and A=01: indicate a checksum error (block was read incorrectly).

Z=0 and A=02: indicate a memory error (block read incorrectly due to bad memory in read buffer area or buffer area extends into ROM or dedicated address area).

Before BLKIN is called the cassette must be in sync and at speed. The buffer starting address must also be in locations $7E and $7F. The program should establish a buffer area large enough to hold the largest block expected to be read. If you are unsure of the block size, set up a read buffer area of 255 bytes, the largest block size possible.

Upon returning, BLKIN will return the block type code and block length in address $7C and $7D. The read buffer area contains the data read from tape. Also returned is the status: an indication of whether the block was read successfully. A successful read is indicated by the Z bit of the CC register being set. If it is set, the program can continue. If the Z bit is clear, the program should display a message to tell the operater it read a block incorrectly. A simple program at this point would have to be run over again. A more sophisticated program would tell the operator to position the tape to the beginning of the file for a second attempt to read the tape.

Listing 8-15 is a subroutine that preserves the registers (except A and CC) and calls BLKIN. The status is returned in the A and CC registers.

```
6600 *SUBROUTINE NAME: RBLOK
6610 *THIS WILL READ A BLOCK IF THE CASSETTE IS
6620 *IN SYNC. 7E+7F=BUFFER ADDR. UPON RETURNING:
6630 *7D=BLOCK LENGTH, 7C=BLOCK TYPE, AND THE
6640 *A REG AND Z BIT INDICATE READ STATUS.
6650 *14 BYTES OF STACK AREA ARE REQUIRED.
6660 ********************************************
6670 RBLOK    PSHS B,X,DP     SAVE REGS
6680          CLRA            GET 00
6690          TFR A,DP        CLEAR DP REG
6700          JSR  [$A006]    CALL BLKIN
6710          PULS B,X,DP     RESTORE REGS
6720          RTS             RETURN
6730 ********************************************
```

Listing 8-15 The RBLOK Subroutine.

Listing 8-16 is a demonstration of writing a block to tape and then reading it back. It consists of two programs; the first (BKOUT) writes the contents of a 100-byte buffer on tape, and the second (RDBLK) reads the block into a different buffer area.

```
100 *PROGRAM NAME: BKOUT
110 *THIS WILL FILL EACH BYTE OF A 100 BYTE
120 *BUFFER WI A NUMBER FROM 00 TO 63 HEX.
130 *THEN THAT BUFFER IS RECORDED ON TAPE.
140 ********************************************
150          ORG $2600
160 BKOUT    STS BKOLD        SAVE ZBUG S REG
170          LDS #22+BKSTK    NEW STACK
180          CLRA             CLEAR A
190          LDX #BKBUF       GET BUF ADDR
200 BK1      STA ,X+          PUT A IN BUFFER
210          INCA             INC A REG
220          CMPA #$64        DONE?
```

```
100 *PROGRAM NAME: BKOUT
110 *THIS WILL FILL EACH BYTE OF A 100 BYTE
120 *BUFFER WI A NUMBER FROM 00 TO 63 HEX.
130 *THEN THAT BUFFER IS RECORDED ON TAPE.
140 ********************************************
150         ORG $2600
160 BKOUT   STS BKOLD       SAVE ZBUG S REG
170         LDS #22+BKSTK   NEW STACK
180         CLRA            CLEAR A
190         LDX #BKBUF      GET BUF ADDR
200 BK1     STA ,X+         PUT A IN BUFFER
210         INCA            INC A REG
220         CMPA #$64       DONE?
230         BLS BK1         IF NOT,DO AGAIN
240         JSR WLDR        WRITE LEADER
250         LDA #$01        BLOCK TYPE
260         STA $007C       PASS IT
270         LDA #100        BLOCK LENGTH
280         STA $007D       PASS IT
290         LDX #BKBUF      GET BUF ADDR
300         STX $007E       PASS IT
310         JSR WBLOK       WRITE BLOCK
320         JSR MOTOFF      TURN MOTOR OFF
330         LDS BKOLD       GET ZBUG S
340         SWI             RETURN TO ZBUG
350 ********************************************
360 BKOLD   RMB 2           ZBUG S STORAGE
370 BKBUF   RMB 100         BUFFER AREA
380 BKSTK   RMB 22          STACK AREA
390 ********************************************
400 *PROGRAM NAME: RDBLK
410 *THIS WILL READ THE PREVIOUSLY WRITTEN
420 *BLOCK INTO A DIFFERENT BUFFER.
430 ********************************************
440 RDBLK   STS RDOLD       SAVE ZBUG S
450         LDS #16+RDSTK   NEW STACK
460         JSR RLDR        GET IN SYNC
470         LDX #RDBUF      GET BUF ADDR
480         STX $007E       PASS IT
490         JSR RBLOK       READ BLOCK
500         JSR MOTOFF      TURN OFF MOTOR
510         LDS RDOLD       GET ZBUG S
520         SWI             RETURN TO ZBUG
530 ********************************************
540 RDOLD   RMB 2           ZBUG S STORAGE
550 RDSTK   RMB 16          STACK AREA
560 RDBUF   RMB 100         BUFFER AREA
570 ********************************************
5800 *SUBROUTINE NAME: WLDR
5810 *THIS WILL TURN ON THE CASSETTE MOTOR AND
5820 *WRITE A GAP AND LEADER. 14 BYTES OF STACK
5830 *AREA ARE REQUIRED.
5840 ********************************************
5850 WLDR   PSHS D,X,Y,DP,CC SAVE REGS
5860        CLRB            GET 00
5870        TFR B,DP        CLEAR DP REG
5880        JSR [$A00C]     CALL WRTLDR
5890        PULS D,X,Y,DP,CC RESTORE REGS
5900        RTS             RETURN
5910 ********************************************
```

```
*SUBROUTINE NAME: WBLOK
*THIS WILL WRITE HEADER, DATA, OR EOF BLOCK
*ON CASSETTE TAPE. ADDRESSES 7C=BLOCK TYPE;
*7D=BUFFER LENGTH; 7E+7F=BUFFER ADDRESS.
*20 BYTES OF STACK AREA ARE REQUIRED.
*******************************************
WBLOK   PSHS 0,X,Y,DP,CC SAVE REGS
        CLRB             GET 00
        TFR B,DP         CLEAR DP REG
        JSR [$A008]      CALL BLKOUT
        PULS D,X,Y,DP,CC RESTORE REGS
        RTS              RETURN
*******************************************
*SUBROUTINE NAME: MOTOFF
*THIS SUBROUTINE WILL TURN OFF THE CASSETTE
*MOTOR. TWO BYTES OF STACK AREA ARE USED.
*******************************************
MOTOFF  PSHS A,CC        SAVE REGS
        LDA $FF21        GET CONTROL BYTE
        ANDA #$F7        CLEAR BIT 3
        STA $FF21        STORE CONTRL BYTE
        PULS A,CC        RESTORE REGS
        RTS              RETURN
*******************************************
*SUBROUTINE NAME: RLDR
*THIS WILL TURN ON THE CASSETTE MOTOR AND
*USE THE LEADER TO GET IN SYNC FOR READING
*THE FOLLOWING BLOCK. 14 BYTES OF STACK
*AREA ARE USED.
*******************************************
RLDR    PSHS D,X,CC,DP   SAVE REGS
        CLRB             GET 00
        TFR B,0P         CLEAR DP REG
        JSR [$A004]      CALL CSRDON
        PULS D,X,CC,DP   RESTORE REGS
        RTS              RETURN
*******************************************
*SUBROUTINE NAME: RBLOK
*THIS WILL READ A BLOCK IF THE CASSETTE IS
*IN SYNC. 7E+7F=BUFFER ADDR. UPON RETURNING;
*7D=BLOCK LENGTH, 7C=BLOCK TYPE, AND THE
*A REG AND Z BIT INDICATE READ STATUS.
*14 BYTES OF STACK AREA ARE REQUIRED.
*******************************************
RBLOK   PSHS B,X,DP      SAVE REGS
        CLRA             GET 00
        TFR A,DP         CLEAR DP REG
        JSR [$A006]      CALL BLKIN
        PULS B,X,DP      RESTORE REGS
        RTS              RETURN
*******************************************
        END
```

Listing 8-16 The BKOUT and RDBLK Programs.

Assemble the source code with the A/IM/WE/AO command and verify there are no errors. Both programs are now in memory. Go to ZBUG and run the first program by entering GBKOUT. It writes a leader and a data block on tape. Position the tape back to the starting point and run RDBLK. This reads the data into the RDBUF buffer area. Use ZBUG to verify the data was loaded into RDBUF by observing that it is the same as the data in BKBUF.

**Disk I/O**

Disk drives can be connected to the Color Computer; they store and read data to and from 5 1/4 inch floppy disks. The following descriptions apply to the disk drives and controller sold by Radio Shack for the Color Computer.

When disk drives are added to the Color Computer, Disk Extended Color BASIC provides the extra capabilities required to use the disks. When the disk controller is plugged into the ROM module slot the added programs reside in ROM from address $C000 to $DFFF. EDTASM+ and disks can not be used at the same time, because only one can be plugged in to the Color Computer at a time. You can use EDTASM+ to generate object code on tape, and then connect the disk drives and load the object code from tape. First let's look at the manner in which data is organized on disk.

The disks are coated with a film of magnetic material so data can be recorded on it much like the way data is recorded on magnetic tape. Data is recorded on tape along its length. Data is recorded on disk along the length of a narrow circular path on one side of the disk as the disk is rotated by the disk drive. On each disk are 35 circular paths known as **tracks**. Each track is identified by a number, from 0 to 34. The track organization on a disk is illustrated in Fig. 8-13. The small hole in the disk, seen in Fig. 8-13, is used to identify the starting point of the tracks. Track 0 is the outermost circular path, and higher-numbered tracks are positioned inward. The tracks are concentric circles, not a spiral.

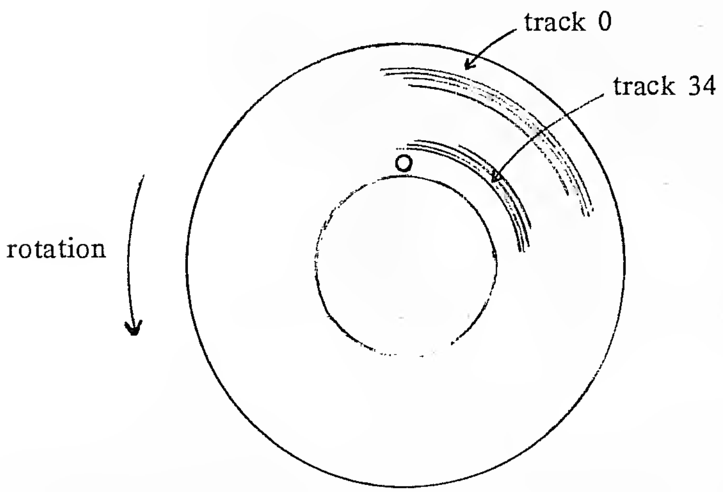


Fig. 8-13 Tracks On a Disk.

Each track is divided into 18 **sectors** of equal length. Each sector is identified by its number, from 1 to 18. Each sector contains 256 bytes of data. A sector is accessed by positioning a **read/write head** over the desired track and waiting until the desired sector passes by. A buffer of 256 bytes can be written to or read from a sector as it passes by the read/write head. The act of moving the read/write head to a track is called **seeking**.

The sectors are numbered by the DSKINI BASIC command, normally set up with a **skip factor** of four. This causes the sectors to be labelled as shown in Fig. 8-14. The skip factor of four causes four sectors to be between consecutively numbered sectors. In Fig. 8-14 sectors 12, 5, 16, and 9 are between sectors 1 and 2.

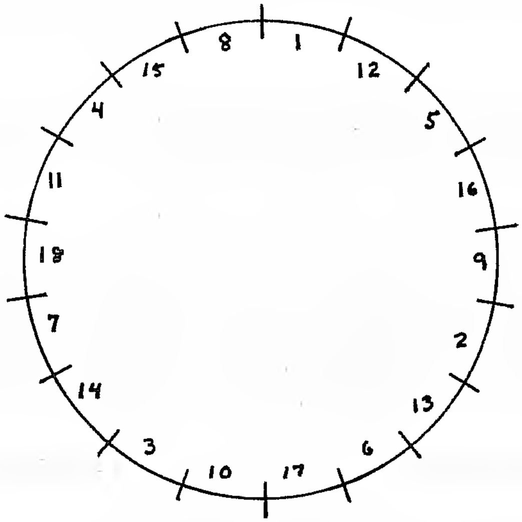


Fig. 8-14 Sector Numbering.

Tracks 0 - 16 and 18 - 34 store data. Track 17 contains the **disk directory**. The disk directory is used by BASIC to keep track of where data has and has not been recorded on the disk. BASIC assigns disk space to a data or program file in **granules** of 9 sectors, one half a track. A BASIC data file that requires 12 sectors would be assigned two granules, or 18 sectors. There are 68 granules, numbered from 0 - 67. Granule 0 is composed of sectors 0 - 9, the first half of track 0. Granule 1 is located in sectors 10 - 18, the second half of track 0. This assignment scheme continues up to granule 67 which occupies the second half of track 34.

The disk directory is composed of two sections: the **file allocation table** and the **directory entries**. The file allocation table is stored in sector 2, and the directory entries are stored in sectors 3 - 11 of track 17. Sectors 12 - 18 of track 17 are not used. The directory entries contain up to 72 entries, each 32 bytes long. Each directory entry describes a different file on the disk. The format of an entry is shown in Fig. 8-15. The bytes in an entry are described as follows:

0-7 - File name, left justified and blank filled. If byte 0 = 00; a previous entry has been deleted and this entry is available for use. If byte 0 = FF; this entry and all following entries have not been used and are available for use.

8-10 - File name extension, left justified and blank filled.

11 - File type code: 00=BASIC program file, 01=BASIC data file, 02=object code program file, 03=text editor source code file.

12 - ASCII flag: 00 = binay format, FF = ASCII format.

13 - The number of the first granule assigned to this file. May contain from 00 to $43.

14-15 - The number of bytes stored in the last sector of this file.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16        32
nn nn nn nn nn nn nn nn xx xx xx tt aa gg bb bb     unused
```

Fig. 8-15 Format of a Directory Entry.

The file allocation table is composed of an entry for each granule. Each entry is one byte long, so bytes 0 - 67 contain the granule information and the rest of the bytes are cleared and not used. Byte 0 contains information about granule 0, byte 1 about granule 1, up to byte 67, which describes granule 67. The contents of each byte are interpreted as follows:

If byte = FF, that granule is not currently part of a file and is available.

If byte = 00 - $43, that granule is part of a file and the contents of this byte is the number of the next granule in the file.

If byte = $C0 - $C9, that granule is the last granule of a file. Represented in bits 0 - 5 is the number of sectors in this granule in use.

The directory entries and file allocation table are used to determine what sectors and/or granules are in use. For more detailed information, see Color Computer Disk System, the manual that comes with the Radio Shack disk drives.

■ DSKCON

Function: This subroutine provides for writing and reading data to and from disk.

Address: [$C004] S Stack Use: 25 bytes

Registers Modified: CC.

Pass Parameters: The DP register must contain 00. The parameters are passed in a table that the contents of addresses $C006 and $C007 point to.

[$C006:$C007] = Operation code where:
  00 = move read/write head to track 0;
  01 = no operation;
  02 = read data from sector into memory;
  03 = write data from memory to sector;
[$C006:$C007] +1 = Disk drive number (00 - 03);
[$C006:$C007] +2 = track number (00 to 22 hex);
[$C006:$C007] +3 = sector number (01 to 12 hex);
[$C006:$C007] +4, +5 = starting address of buffer area.

Return Values: A status byte is returned in memory address [$C006:C007] +6. Bits 2 - 7 indicate the result of attempting a disk operation. If this byte is 00 at the end of an operation, the operation was performed successfully. The meanings of each bit, if set, are:

Bit 7 - Disk drive is not ready. This should never occur because this signal is not used.
Bit 6 - Write protect error. One tried to write on a write protected disk.
Bit 5 - Write fault. An error was detected during a write operation.
Bit 4 - Seek error or sector not found. Also occurs if the disk was not initialized with the DSKINI command or if one tries to access a non-existant sector.
Bit 3 - CRC error. Indicates data was not read correctly from the disk.
Bit 2 - Lost data during a read or write because the MPU didn't respond to the disk's data request soon enough.

Any of the above errors can also occur if something is wrong with the disk or disk drive.

The starting address of DSKCON is located in addresses $C004 and $C005. It can be called with extended indirect addressing.

```
JSR  [$C004]     or     JSR  $D66C
```

One can also read the value (typically $D66C) from $C004 and $C005 and use extended addressing.

The parameters are passed to DSKCON in a table whose starting address is contained in addresses $C006 and $C007. This table typically starts at address $EA. At location $EA (first byte) is the desired disk operation code. At location $EB (second byte) is the desired disk drive number. At location $EC (third byte) is the track number. At location $ED (fourth byte) is the desired sector number. In the fifth and sixth bytes, locations $EE and $EF, is put the starting address of the buffer area. Now DSKCON can be called. When writing, the data in the buffer area (256 bytes) is written on the sector of the selected track. When reading, 256 bytes are read from the sector of the selected track into the buffer area. When DSKCON returns, the return status is in location $F0 or [C006:C007] +6. If its contents are 00, the operation was performed successfully.

When DSKCON is called, it turns on the disk drive motor and performs the operation passed to it. However, upon returning it does not turn the motor off. This is done by clearing the contents of dedicated address $FF40 with the instruction: CLR $FF40. One does not have to turn off the disk motor after every operation. If reading in several sectors, the motor only needs to be turned off after reading in the last desired sector. If there is a long time period between disk operations it is best to turn off the motor to reduce wear.

Since DSKCON does not preserve the contents of the CC register, it may be called from within a subroutine that saves and restores the CC register. This subroutine in Listing 8-17 also ensures the DP register is clear before calling DSKCON.

```
6800 *SUBROUTINE NAME: DISK
6810 *THIS SUBROUTINE WILL CALL DSKCON TO PROVIDE
6820 *ACCESS TO A DISK. 30 BYTES STACK AREA ARE USED
6830 ***********************************************
6840 DISK    PSHS A,CC,DP    SAVE REGS
6850         CLRA            GET 00
6860         TFR A,DP        CLEAR DP REG
6870         JSR [$C004]     CALL DSKCON
6880         PULS A,CC,DP    RESTORE REGS
6890         RTS             RETURN
6900 ***********************************************
```

Listing 8-17 The DISK Subroutine.

Listing 8-18, DGRAN, is a demonstration of using the DISK subroutine. It reads each sector of each track on drive 0 and checks the resulting status. When a bad status is detected, that track/sector number is displayed on the screen. The program continues in this fashion to the last track. The subroutine ASCNUM used is similar to the PHEX program of Chapter 7.

```
100 *PROGRAM NAME: DGRAN
110 *THIS PROGRAM WILL READ EVERY SECTOR ON DRIVE
120 * 0 AND DISPLAY TRACK/SECTORS THAT ARE
130 *DEFECTIVE.
140 ***********************************************
150          ORG $2800       LOAD ADDR
160 DGRAN    STS DGOS        SAVE S
170          LDS #32+DGSTK   NEW STACK
180          JSR CLS         CLEAR SCREEN
190          LDY $C006       GET TABLE ADDR
200          LDD #$0001      TRK 0, SECT 1
210 DG1      STD 2,Y         PUT IN TABLE
220          LDU #$0200      READ, DRIVE 0
230          STU ,Y          PUT IN TABLE
240          LDU #DGBF       GET BUF ADDR
250          STU 4,Y         PUT IN TABLE
260          JSR DISK        READ DISK
270          TST 6,Y         ERRORS ?
```

```
 280          BEQ DG2         BRANCH IF NONE
 290          LDX #DGETX      GET MESS ADDR
 300          STA ASBYTE      PASS TO ASCNUM
 310          JSR ASCNUM      CONVERT TO ASCII
 320          LDU ASRES       GET RESULT
 330          STU 10,X        PUT IN MESSAGE
 340          STB ASBYTE      PASS TO ASCNUM
 350          JSR ASCNUM      CONVERT TO ASCII
 360          LDU ASRES       GET RESULT
 370          STU 20,X        PUT IN MESSAGE
 380          JSR DSPLAY      DISPLAY MESSG
 390 DG2      INCB            NEXT SECTOR
 400          CMPB #18        18 OR LESS ?
 410          BLS DG3         YES, BRANCH
 420          LDB #$01        SECTOR TO ONE
 430          INCA            NEXT TRACK
 440 DG3      CMPA #34        LAST TRACK ?
 450          BLS DG1         NO- DO AGAIN
 470 DGEND    LDS DGOS        GET S
 480          RTS             RETURN TO BASIC
 490 **********************************************
 500 DGOS     RMB 2           BASIC S STORAGE
 510 DGSTK    RMB 32          STACK AREA
 520 DGETX    FCC /BAD TRACK     SECTOR   /
 530          FDB $0D22       CR/LF AND END
 540 DGBF     RMB 256         BUFFER AREA
 550 **********************************************
5000 *SUBROUTINE NAME: CLS
5010 *THIS SUBROUTINE WILL CLEAR THE TEXT SCREEN
5020 *AND RESET THE DISPLAY POINTER. 6 BYTES OF
5030 *THE S STACK ARE USED.
5040 **********************************************
5050 CLS      PSHS B,X,CC     SAVE REGS
5060          JSR $A928       CALL CLSCRN
5070          PULS B,X,CC     RESTORE REGS
5080          RTS             RETURN
5090 **********************************************
5500 *SUBROUTINE NAME: DSPLAY
5510 *THIS SUBROUTINE WILL DISPLAY THE TEXT STRING
5520 *(ASCII CODES) THAT THE X REGISTER POINTS TO
5530 *STARTING AT THE POSITION SPECIFIED BY THE
5540 *DISPLAY POINTER. A CODE OF 0D IN THE STRING
5550 *WILL CAUSE A CR/LF. A CODE OF 22 (")
5560 *INDICATES THE END OF A STRING TO DISPLAY. 20
5570 *BYTES OF STACK AREA ARE REQUIRED.
5580 **********************************************
5590 DSPLAY   PSHS D,X,U,CC,DP SAVE REGS
5600          CLRB            GET 00
5610          TFR B,DP        CLEAR DP REG
5620          LEAX -1,X       ADJUST POINTER
5630          JSR $B99C       CALL DISPL
5640          PULS D,X,U,CC,DP RESTORE REGS
5650          RTS             RETURN
5660 **********************************************
6800 *SUBROUTINE NAME: DISK
6810 *THIS SUBROUTINE WILL CALL DSKCON TO PROVIDE
6820 *ACCESS TO A DISK. 30 BYTES OF STACK ARE USED.
6830 **********************************************
6840 DISK     PSHS A,CC,DP    SAVE REGS
6850          CLRA            GET 00
```

```
6860          TFR A,DP       CLEAR DP REG
6870          JSR [$C004]    CALL DSKCON
6880          PULS A,CC,DP   RESTORE REGS
6890          RTS            RETURN
6900 ********************************************
7000 *SUBROUTINE NAME: ASCNUM
7010 *THIS WILL GENERATE THE ASCII CODES OF TWO
7020 *HEX DIGITS IN BYTE ASBYTE. THE TWO CODES ARE
7030 *PUT IN ASRES.
7040 ********************************************
7050 ASCNUM   PSHS D         SAVE REGS
7060          LDB ASBYTE     GET BYTE
7070          LDA #$10       MULTIPLIER VALUE
7080          MUL            SHIFT MS NIBBLE TO A
7090          CMPA #$09      GREATER THAN 9 ?
7100          BHI ASC1       YES, JUMP
7110          ADDA #48       MAKE ASCII
7120          BRA ASC2       CONTINUE
7130 ASC1     ADDA #55       MAKE ASCII
7140 ASC2     STA ASRES      STORE RESULT
7150          LDA ASBYTE     GET BYTE
7160          ANDA #$0F      GET LS NIBBLE
7170          CMPA #$09      GREATER THAN 9 ?
7180          BHI ASC3       YES, JUMP
7190          ADDA #48       MAKE ASCII
7200          BRA ASC4       CONTINUE
7210 ASC3     ADDA #55       MAKE ASCII
7220 ASC4     STA 1+ASRES    STORE RESULT
7230          PULS D,PC      RESTORE + RETURN
7240 ********************************************
7250 ASBYTE   RMB 1          DECODE BYTE
7260 ASRES    RMB 2          RESULTING CODE
7270 ********************************************
7280          END $2800      EXEC ADDR
```

Listing 8-18 The DRGRAN Program.

Assemble the source code on tape with the command: A DGRAN/WE, and verify that there are no errors. Turn off the Color Computer, unplug the EDTASM+ module and connect the disk drives. Turn everything on and load DGRAN from tape with the CLOADM command. The program is run with the EXEC command.

[illegible]

# CHAPTER 9

# Internal Control and Graphics

The Color Computer has more major components, serving such purposes as input and output, video display generation, interrupt generation, and internal operation control. Each component is controlled through dedicated addresses from the MPU. Fig. 9-1 is a simplified block diagram of the major components in the Color Computer. The new components are the **SAM, PIAs,** and the **VDG**. The SAM, or **synchronous address multiplexer**, synchronizes the operation of the other components so they all work in harmony. The VDG, or **video display generator**, generates a text or graphics display. Through the PIAs, or **peripheral interface adapters**, one controls external devices: input or output data; the VDG; and some of the interrupt sources.

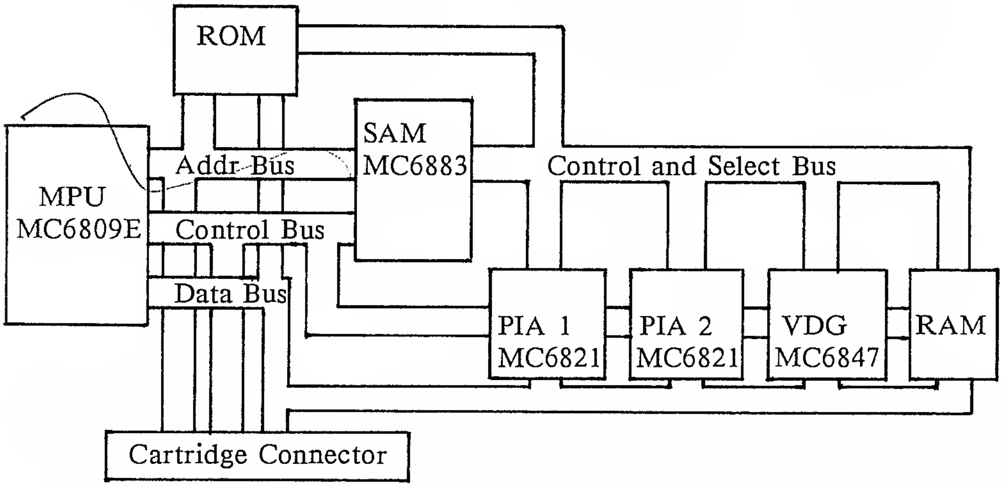


Fig. 9-1 Block Diagram of the Color Computer.

## SYNCHRONOUS ADDRESS MULTIPLEXER

The SAM is the general controller of all the components in the Color Computer. It is a large scale integrated circuit: a Motorola MC6883. It synchronizes the operation of the various components so they work together, decodes addresses, and controls the operating modes of other components. Synchronizing the major components isn't covered because it is primarily a hardware function and not related to programming; this operation is transparent to a programmer.

### Controlling the SAM

The operation of the SAM is directed through 16 control bits. The bits are set or cleared by writing into dedicated addresses. No value is stored in these dedicated addresses; the act of writing into one of them causes a control bit within the SAM to be set or cleared. The SAM control bits, their dedicated addresses, and their uses, are shown in Table 9-1. A control bit is cleared by writing into its even-numbered dedicated address and set by writing into the odd-numbered address. Following Table 9-1 is a description of each control bit.

| Control Bit | Function | Dedicated Address | |
|---|---|---|---|
| | | Set | Clear |
| TY | Memory map mode | FFDF | FFDE |
| M1 | Memory | FFDD | FFDC |
| M0 | size | FFDB | FFDA |
| R1 | MPU | FFD9 | FFD8 |
| R0 | cycle rate | FFD7 | FFD6 |
| P1 | Page Number | FFD5 | FFD4 |
| F6 | | FFD3 | FFD2 |
| F5 | | FFD1 | FFD0 |
| F4 | Video display | FFCF | FFCE |
| F3 | starting address | FFCD | FFCC |
| F2 | | FFCB | FFCA |
| F1 | | FFC9 | FFC8 |
| F0 | | FFC7 | FFC6 |
| V2 | Video display | FFC5 | FFC4 |
| V1 | mode | FFC3 | FFC2 |
| V0 | | FFC1 | FFC0 |

Table 9-1 SAM Control Bits.

■ Memory Map Mode - This bit, TY, is cleared by BASIC when the computer is turned on. The TY bit tells the SAM how to map memory. When it is clear, RAM memory is available from addresses 0000 - $7FFF in a computer with 32K or 64K RAM. In a computer with 16K RAM, the RAM addresses extend from 0000 - $3FFF. Addresses $8000 - $BFFF access the internal ROM, which

[illegible]

contains Color BASIC and Extended Color BASIC. Addresses $C000 - $FEFF access the ROM plugged into the ROM cartridge slot. Addresses $FF00 - $FFFF are dedicated addresses. Thus when the TY bit is clear, RAM is available in the first half of memory and internal and/or external ROM is available in the second half of memory.

When the TY bit is set the SAM directs all addresses from the MPU to RAM, except for the dedicated addresses $FF00 through $FFFF. For this mode to operate correctly the older Color Computer with 64K must be modified to allow accessing of RAM addresses $8000 through $FEFF. A letter entitled "Color Computer Enhancements" by Robert Brooks in the September, 1982 issue of **80 Micro** describes a method to implement this modification. The Color Computer 2 already has this modification. All descriptions of the SAM operation with the TY bit set assume this modification has been installed. When TY is set, BASIC and cartridge ROM are not avaliable, but a Color Computer with 64K RAM can use nearly all of it, instead of just the first 32K. In a 64K computer longer assembly language programs can reside in memory but ROM is not available. Memory maps for TY being set and cleared are shown in Fig. 9-2.

SAMs produced before January 1, 1983 do not operate when TY and R0 or R1 are set (see MPU Clock Rate description). The date of manufacture is marked on the MC6883 as YYWW where YY is the year and WW is the week. The Color Computer 2 contains the newer SAM.

When the TY bit is set, only the amount of RAM in the computer is available. Thus a 16K computer will have 16K of RAM and no ROM available when the TY bit is set.

| TY Clear | Address | TY Set |
|---|---|---|
| RAM | 0000 | RAM |
| Extended Color BASIC | 8000 | |
| Color BASIC | A000 | |
| External ROM (cartridge) | C000 | |
| Dedicated Addresses | FF00 FFFF | Dedicated Addresses |

Fig. 9-2 Effect of TY Bit on Memory Map.

■ Memory Size - The memory size bits, M0 and M1, tell the SAM what type of RAM integrated circuits are in the computer. The three different types correspond to the three RAM sizes: 4K, 16K, and 32/64K. The M0 and M1 bits are set or cleared by BASIC to the value corresponding to memory size when the computer is turned on. Thereafter their states should not be changed or data may be lost from RAM. The states of M0 and M1 which correspond to the different memory sizes are shown in Table 9-2. The M0 and M1 bits do not specify the amount of RAM available, but tell the SAM how to use the type of memory integrated circuits in the computer.

| M1 | M0 | Memory Size | |
|---|---|---|---|
| 0 | 0 | 4K | |
| 0 | 1 | 16K | |
| 1 | 0 | 32/64K | - 64 K dynamic |
| 1 | 1 | not used | 64 K static |

Table 9-2 Memory Size Control Bits.

■ MPU Clock Rate - Control bits R0 and R1 control the MPU clock signals, E and Q, generated by the SAM. Usually both bits are clear, setting a 0.89 megahertz (million cycles per second) MPU clock rate. The clock signals step the MPU through its sequences and so control its speed of operation. The settings and results of the R0 and R1 bits are shown in Table 9-3. The clock rate is also dependent on what addresses are being accessed by the MPU.

| | | Clock Rate While Addressing | | | |
|---|---|---|---|---|---|
| R1 | R0 | 0-7FFF | 8000-FEFF | FF00-FF1F | FF20-FFFF |
| 0 | 0 | 0.89 | 0.89 | 0.89 | 0.89 |
| 0 | 1 | 0.89 | 1.78 | 0.89 | 1.78 |
| 1 | X | 1.78 | 1.78 | 1.78 | 1.78 |

Table 9-3 Clock Rates in Megahertz.

As you can see, setting the R0 bit doubles the operating speed of the MPU while it is operating in ROM. When running at this higher speed the computer will execute BASIC programs approximately 40 percent faster. The execution speed is not doubled because the BASIC statements are in RAM where the clock rate is at the slow speed. Assembly language programs residing in the lower 32K will run at the normal speed when R0 is set. The higher speed should be used with caution however, because this cycle rate is beyond the specifications of the MC6809E and it may not run properly. If you want the MPU to run reliably at the higher speed, the MPU should be replaced with an MC68B09E; the high speed version of the MC6809E.

Setting R1 sets the high speed mode while accessing all addresses. This will require the RAM and PIAs to be capable of this high speed. The high speed PIA is the MC68B21.

When ROM I/O subroutines are used to output or input data to or from external devices the clock rate must be set to its normal rate: R1 and R0 must be cleared for the subroutines to work properly. The MPU execution rate affects the rate at which data is output or input by the subroutines. Also some ROM modules, such as EDTASM+, will not work when the clock rate is set high.

■ Page Number - The page number bit, P1, is cleared by BASIC when the computer is turned on. The effect of P1 set can only be used by a 64K Color Computer. A 64K RAM is considered as two 32K pages. When the page number bit is clear page 0 is the first 32K and page 1 is the upper 32K. With TY clear (RAM/ROM mode) and the page number bit clear the first 32K of RAM occupies addresses 0000 through $7FFF and ROM extends upward from there. Setting the page number bit results in the upper 32K of RAM, which was not accessable with TY clear, being accessed by addresses 0000 through $7FFF. This allows you to have two BASIC programs in RAM at the same time; one in page 0 and the other in page 1.

With TY set (all RAM mode) the page number bit controls which page of 64K is the lower half: that accessed by addresses 0000 through $7FFF. The other page is then the upper half.

■ Video Display Starting Address - The video display starting address, bits F0 through F6, specify the starting address of the video (screen) buffer area. The seven bits are used to construct a 16-bit address as seen below:

```
F6 F5 F4 F3 F2 F1 F0  0    0 0 0 0 0 0 0 0
```

Bits F0 through F6 make up the upper seven bits and the other bits are clear. When the computer is turned on, BASIC sets bit F1 and clears the rest, giving a buffer starting address of $0400. When graphics are displayed using BASIC the F0 bit is set so the graphics video buffer starts at address $0600. One can specify a video buffer at any address in RAM that is a multiple of $200 with the video display starting address bits. Using several video buffers in memory at the same time, one can very quickly change a display just by changing the states of the F control bits to point to the desired buffer. The length of the video buffer depends on the display mode being used. (See the next item.)

■ Video Display Mode - The video display mode bits, V0, V1, and V2, are used in conjunction with the video display generator. These bits control the size of the buffer area the SAM sends to the VDG for conversion to a video signal, which is then sent to the television set. The SAM sends the buffer contents to the VDG 60 times a second, the rate at which the picture is continually repainted on the TV screen. The valid settings of the video display mode bits, the buffer size, and the display modes they use are shown in Table 9-4.

| V2 | V1 | V0 | Buffer Size | Display Modes |
|---|---|---|---|---|
| 0 | 0 | 0 | 512 | AI,SG4,SG6 |
| 0 | 0 | 1 | 1024 | G1C, G1R |
| 0 | 1 | 0 | 2048 | G2C, SG8 |
| 0 | 1 | 1 | 1536 | G2R |
| 1 | 0 | 0 | 3072 | G3C, SG12 |
| 1 | 0 | 1 | 3072 | G3R |
| 1 | 1 | 0 | 6144 | G6R,G6C,SG24 |
| 1 | 1 | 1 | Not Used | |

Table 9-4 SAM Video Display Modes.

The buffer sizes are listed in decimal. The display modes will be described in the Video Display Generator section of this chapter. There are more details to be described concerning directing the VDG to generate the desired display mode.

**Decoding Addresses**

A major SAM task affecting programs is how it decodes addresses. The MPU sends a 16-bit address to the SAM, which inspects and decodes it. Upon decoding the address, the SAM selects the appropriate component to which data is to be written or from which data is to be read. The components that may be addressed are RAM, ROM, the VDG, the PIAs, ROM in the cartridge slot, and the SAM itself. However, the state of the SAM control bit, TY, will affect this decoding process. Following is a description of the addresses whose decoding is affected by the state of the TY bit.

■ Decoding with TY Clear - When the TY bit is clear, the addresses are decoded as shown in Fig. 9-2. Upon closer inspection, one will find that certain locations can be accessed with more than one address.

The interrupt vector table is accessed with addresses $FFF0 through $FFFF. This table actually resides in BASIC ROM addresses $BFF0 - $BFFF. One can display both tables and see that they are the same. Upon receiving an address from $FFF0 to $FFFF the SAM will decode it as though it were a ROM address of $BFF0 through $BFFF.

PIA 1 is accessed via four dedicated addresses, $FF00 through $FF03. PIA 2 is accessed via four dedicated addresses, $FF20 through $FF23. The SAM, with TY clear, also decodes other addresses as PIA dedicated addresses. The others (redundant addresses) work just as well as the primary dedicated addresses, and are shown in Table 9-5.

| PIA | Primary Dedicated Address | Redundant Dedicated Addresses |
|---|---|---|
| 1 | FF00 | FF04,FF08,FF0C,FF10,FF14,FF18,FF1C |
|  | FF01 | FF05,FF09,FF0D,FF11,FF15,FF19,FF1D |
|  | FF02 | FF06,FF0A,FF0E,FF12,FF16,FF1A,FF1E |
|  | FF03 | FF07,FF0B,FF0F,FF13,FF17,FF1B,FF1F |
| 2 | FF20 | FF24,FF28,FF2C,FF30,FF34,FF38,FF3C |
|  | FF21 | FF25,FF29,FF2D,FF31,FF35,FF39,FF3D |
|  | FF22 | FF26,FF2A,FF2E,FF32,FF36,FF3A,FF3E |
|  | FF23 | FF27,FF2B,FF2F,FF33,FF37,FF3B,FF3F |

Table 9-5 PIA Primary and Redundant Addresses (TY=0).

The disk controller is accessed through five dedicated addresses; $FF40 and $FF48 - $FF4B. The SAM, with TY clear, also decodes other addresses as disk controller dedicated addresses. The redundant addresses, which work just as well as the primary dedicated addresses, are shown in Table 9-6.

| Primary Dedicated Addressed | Redundant Dedicated Addresses |
|---|---|
| FF40 | FF41 - FF47 and FF50 - FF57 |
| FF48 | FF4C, FF58, FF5C |
| FF49 | FF4D, FF59, FF5D |
| FF4A | FF4E, FF5A, FF5E |
| FF4B | FF4F, FF5B, FF5F |

Table 9-6 Disk Controller Primary and Redundant Addresses (TY=0).

■ Decoding With TY Set - When the TY bit is set, the addresses are decoded as shown in Fig. 9-2. The SAM decodes an address as a RAM or primary dedicated address.

The interrupt vector table can be found only at addresses $FFF0 - $FFFF. However, the SAM is still getting their contents from BASIC ROM addresses $BFF0 - $BFFF. The PIAs can be accessed only through their primary dedicated addresses. The primary dedicated addresses for PIAs 1 and 2 are shown in Table 9-5. The disk controller can be accessed only through its primary dedicated address, as shown in Table 9-6. There are no redundant addresses when the TY SAM control bit is set.

## PERIPHERAL INTERFACE ADAPTER

It is through the two peripheral interface adapters, PIAs 1 and 2, that the MPU communicates with external devices: the keyboard, joysticks, cassette, and printer. One PIA is also used to generate sound. Each PIA is a Motorola MC6821 large scale integrated circuit.

A PIA is composed of two halves that are almost identical. One half is the A side and the other is the B side. Each side can input or output a byte of data and send an interrupt to the MPU. Since the Color Computer has two PIAs, there are four data paths to input and output data. Fig. 9-3 is a simplified diagram depicting a PIA with its data paths and buses.

It is via the control bus that the PIA sends an interrupt to the MPU. The control & select bus is used to select a particular PIA and also its A or B side. A byte is transferred between the MPU and a PIA over the data bus. On the right side of the PIA are the data paths, connected to other devices. Each side of a PIA has its own data path of eight bits. In addition, the A side has two signals - CA1 and CA2, and the B side has CB1 and CB2, which are also connected to other components. Data is sent or received on a data path in a parallel fashion. A byte sent out on the A side would send the individual bits 0 - 7 out on signal lines PA0 - PA7, respectively. Each PIA contains within it six registers, three registers in each side. This can be seen in Fig. 9-3.

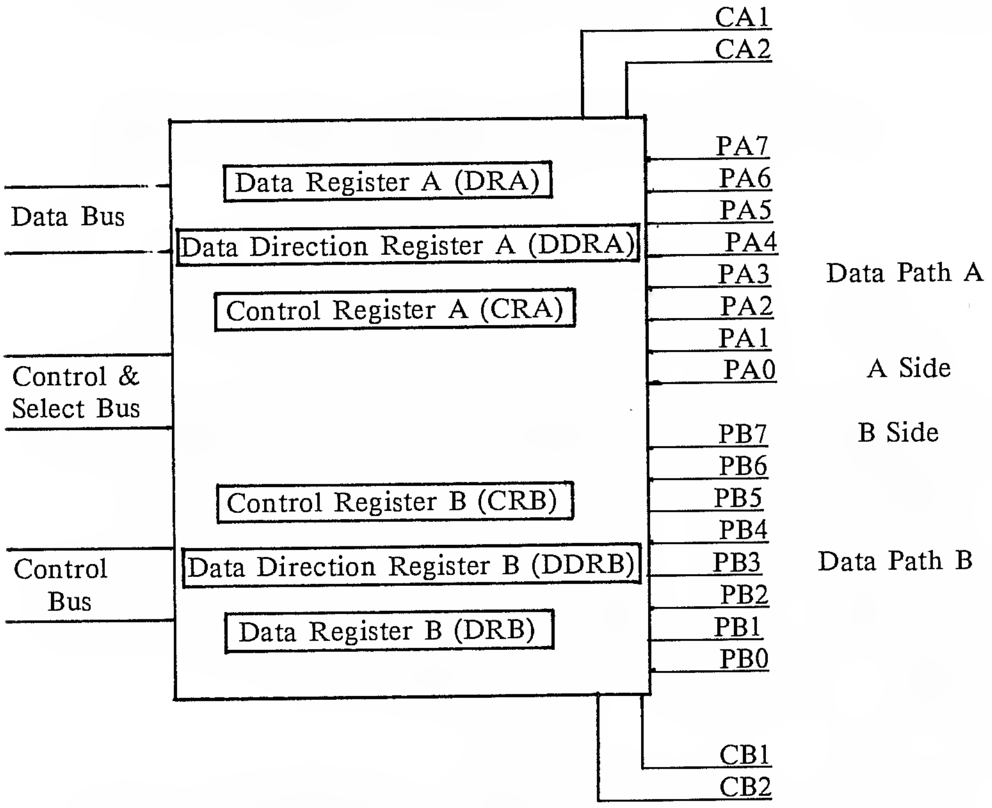


Fig. 9-3 Block Diagram of a PIA.

Each side has a complete set of three registers, so each side can operate independently. One register is the **data register** (DRA or DRB), through which data is sent from the MPU to the data path or from the data path to the MPU. The contents of the **data direction register** (DDRA or DDRB) control the direction the data will flow through the individual bits of the data register. Remember that in and out are referenced to the MPU; out is from the MPU and in is toward the MPU.

The contents of the **control register** (CRA or CRB) control the operation of the CA1, or CB1, and CA2, or CB2 signals, and whether the data register or data direction register can be accessed through a dedicated address by the MPU. The operation of only the A side registers is presented since they generally operate the same as the B side registers. Only the CRB operates slightly differently from the CRA.

## Data Register (DRA or DRB)

The DRA, an eight-bit register, is accessed from the MPU through its dedicated address if bit 2 of the CRA is set. When outputting the MPU can store data in the DRA, which will then be outputted over data path A. When inputting the MPU can read data from the DRA, and the contents of the DRA will reflect the data on data path A. It is through the DRA that the MPU can read the states of the bits PA0 - PA7 or send data out on the data lines PA0 - PA7.

Each bit of the DRA can be configured for outputting or inputting. For example, one can configure DRA0 - DRA3 for output and DRA4 - DRA7 for input. However, a bit can be in only the output or input condition. The data direction of the bits of the DRA is controlled by the DDRA.

## Data Direction Register (DDRA or DDRB)

The DDRA, an eight-bit register, is accessed from the MPU through the same dedicated address as the DRA when bit 2 of the CRA is clear. One dedicated address is used to access either the DRA or DDRA, depending on the state of bit 2 of the CRA. The contents of DDRA will configure each bit of the DRA for input or output. Each bit of the DDRA controls the corresponding bit of the DRA: DDRA0 controls DRA0, DDRA1 controls DRA1,

| DDRA=A5 | DRA | Bit Number |
|---|---|---|
| 1 | > | PA7 |
| 0 | < | PA6 |
| 1 | > | PA5 |
| 0 | < | PA4 |
| 0 | < | PA3 |
| 1 | > | PA2 |
| 0 | < | PA1 |
| 1 | > | PA0 |

Fig. 9-4 Configuring the DRA.

etc. If a bit in the DDRA is clear, the corresponding bit in the DRA is configured for inputting. If a bit in the DDRA is set, the corresponding bit in the DRA is configured for outputting. For example, if the DDRA contains $A5, DRA bits 1, 3, 4, and 6 are configured for inputting, and bits 0, 2, 5, and 7 are configured for outputting. This is graphically shown in Fig. 9-4. The right arrow indicates output only and the left arrow indicates input only. As you can see, the MC6821 is quite a versatile component.

**Control Register (CRA or CRB)**

The CRA is the most complicated register, since it controls the largest number of items. The items controlled include the CA1 signal, DRA/DDRA access, and the CA2 signal, and it provides indications of the states of CA1 and CA2. The CRA, an eight-bit register, is accessed via its dedicated address. The MPU can read data from it and write to some of its bits. The format of the CRA or CRB is shown in Fig. 9-5. Each field in the CRA or CRB will be described. All CRA fields act the same as CRB except for the CA2(CB2) Control field.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| Int. Flag CA(B)1 | Int. Flag CA(B)2 | CA2(CB2) Control | | | DDR/DR Access | CA1(CB1) Control | |

Fig. 9-5 PIA Control Register A(B).
Courtesy of Motorola, Inc.

■ CA1(CB1) Control - The CA1 control field controls how the CA1 signal is interpreted by the PIA. The CA1 signal is always configured as an input in the Color Computer. It is changed by writing to the CRA(B).

Bit 1, when clear, causes the CA1 signal to set an active state in the PIA when the signal received by CA1 from another device changes from a high (logical 1) to a low (logical 0).

Bit 1, when set, causes the CA1 signal to set an active state in the PIA when the signal received by CA1 from another device changes from a low (logical 0) to a high (logical 1).

Bit 0, when clear, lets the CA1 active state set bit 7 (CA1 flag) of the CRA. Bit 7 set indicates that CA1 was activated by the required transition.

Bit 0, when set, lets the CA1 active state set bit 7 of the CRA and send an interrupt to the MPU. Bit 0 of the CRA can enable or disable the sending of interrupts generated by CA1.

The above description is equally applicable to the B side of the PIA: the CRB and CB1.

■ DDR/DR Access - Bit 2 of the CRA is used to determine which register, DRA or DDRA, will be accessed via their dedicated address. If bit 2 is clear, DDRA can be accessed. If bit 2 is set, DRA can be accessed. The state of this bit is changed by writing it into the CRA from the MPU. This description is equally applicable to the B side.

■ CA2 Control - The CA2 signal can be configured as an input or an output. In the Color Computer it is used for output only. Bits 3, 4, and 5 of the CRA control the use of CA2 as shown below.

CA2 configured as an input: (Bit 5 = 0)
- Bit 4: Determines the transition that will activate the CA2 input.
  - Bit 4 = 0: Activate upon seeing a high (logical 1) to low (logical 0) transition.
  - Bit 4 = 1: Activate upon seeing a low to high transition.
- Bit 3: Allows enabling or disabling of an interrupt generated by CA2.
  - Bit 3 = 0: Upon activation of CA2, set bit 6 (CA2 flag) of the CRA.
  - Bit 3 = 1: Upon activation of CA2, set bit 6 (CA2 flag) of the CRA and send an interrupt to the MPU.

CA2 configured as an output: (Bit 5 = 1)
- Bit 4 = 1: CA2 outputs a signal corresponding to the state of CRA bit 3.
  - Bit 3 = 0: CA2 outputs a low.
  - Bit 3 = 1: CA2 outputs a high.
- Bit 4 = 0: CA2 outputs a low upon the next MPU read of the DRA. Bit 3 determines what conditions will cause CA2 to resume outputting a high.
  - Bit 3 = 0: CA2 returns to a high upon the next active CA1 transition.
  - Bit 3 = 1: CA2 returns to a high after one MPU clock cycle.

■ CB2 Control - The CB2 signal can be configured as an input or an output. In the Color Computer it is used for output only. Bits 3, 4, and 5 of the CRB control the use of CB2 as shown below.

CB2 configured as an input: (Bit 5 = 0)
- Bit 4: Determines the transition that will activate the CB2 input.
  - Bit 4 = 0: Activate upon seeing a high (logical 1) to low (logical 0) transition.
  - Bit 4 = 1: Activate upon seeing a low to high transition.
- Bit 3: Allows enabling or disabling of an interrupt generated by CA2.
  - Bit 3 = 0: Upon activation of CB2, set bit 6 (CB2 flag) of the CRB.
  - Bit 3 = 1: Upon activation of CB2, set bit 6 (CB2 flag) of the CRB and send an interrupt to the MPU.

CA2 configured as an output: (Bit 5 = 1)
Bit 4 = 1: CB2 outputs a signal corresponding to the state of CRB bit 3.
Bit 3 = 0: CB2 outputs a low.
Bit 3 = 1: CB2 outputs a high.
Bit 4 = 0: CB2 outputs a low upon the next MPU write to the DRB. Bit 3 determines what conditions will cause CB2 to resume outputting a high.
Bit 3 = 0: CB2 returns to a high upon the next active CB1 transition. Bit 7 of the CRB must have been previously cleared.
Bit 3 = 1: CB2 returns to a high after one MPU clock cycle.

## Using the PIAs

A PIA can be configured to perform many ways; in the Color Computer, however, the PIAs are connected to devices that can be used in only limited ways. Therefore, not all the PIA configurations are used. Presented in this section are more details on using the PIAs and how they apply to the Color Computer. Chapter 10 describes the devices and how they are connected to the PIAs.

■ PIA Dedicated Addresses - Each PIA can be accessed by four primary dedicated addresses and their redundant addresses if the TY bit of the SAM is clear. (See Table 9-5 for all the possible addresses.) The primary dedicated addresses for each PIA register are shonw in Table 9-7.

| Address | PIA Registers |
|---|---|
| FF00 | DRA and DDRA of PIA 1 |
| FF01 | CRA of PIA 1 |
| FF02 | DRB and DDRB of PIA 1 |
| FF03 | CRB of PIA 1 |
| FF20 | DRA and DDRA of PIA 2 |
| FF21 | CRA of PIA 2 |
| FF22 | DRB and DDRB of PIA 2 |
| FF23 | CRB of PIA 2 |

Table 9-7 Primary Dedicated Addresses of the PIA Registers.

■ Initializing the PIAs - The PIAs are initialized by BASIC when the computer is turned on. This involves setting up DDRA(B), and CRA(B) for each PIA. The PIAs are first initialized by the RESET signal, activated at turn on or when the Reset button is pushed. A RESET clears all the registers in the PIAs. In this state, all the data register bits are configured to output and interrupts are disabled.

BASIC sets up the data direction and control registers of each PIA register as shown in Table 9-8.

| PIA Register | Contents |
|---|---|
| DDRA of PIA 1 | 00 |
| CRA of PIA 1 | 34 |
| DDRB of PIA 1 | FF |
| CRB of PIA 1 | 35 |
| DDRA of PIA 2 | FE |
| CRA of PIA 2 | 30 |
| DDRB of PIA 2 | F8 |
| CRB of PIA 2 | 30 |

Table 9-8 PIA Registers Initailized by BASIC.

One can see, for example, that the A side of PIA 1 has its DRA set up to input only, and the CRA is set up so CA1 interrupts are disabled, the DRA can be accessed, and CA2 is configured to output a logical 0. In the Color Computer, the only CA2(CB2) configuration which is used is outputting a 0 or 1 bit.

■ The Interrupt Flags - The interrupt flags are bits 6 and 7 of the control registers. Bit 7 set in CRA indicates that the CA1 input has been activated. Bit 6 set in the CRA indicates the CA2 input has been activated. You can not change their state by writing to the CRA(CRB). You should never see bit 6 set because the only configuration of CA2(CB2) usable in the Color Computer is the output mode.

When bit 7 of CRA(CRB) is set, it must be cleared for the PIA to be able to detect the next activation of CA1(CB1). If CA1(CB1) interrupts are enabled (bit 0 of the control register set) the interrupt signal will be sent to the MPU as long as bit 7 is set. Therefore the interrupt handler program must clear bit 7 to acknowledge to the PIA that the interrupt was received. The interrupt flag is cleared by reading the contents of the respective DRA(DRB). Upon reading the data register, its contents will be in an MPU register and bits 6 and 7 in the control register are cleared.

## VIDEO DISPLAY GENERATOR

The video display generator, or VDG, is the component which generates a video signal sent to a television set to produce the picture. The VDG has a number of display modes; six display text or semi-graphics and the others display graphics. The display modes are selected by setting the required control bits in the SAM and by selecting the VDG operating mode.

### Selecting the Display Mode

A display mode is selected by setting the video display starting address, control bits F0 - F6; and the video display mode, control bits V0 - V2, all in the SAM. Also, the VDG operating mode, controlled by five VDG control bits, must be set.

The video display starting address is normally set by BASIC to $0400, bit F1 set, for displaying text. Bits F0 - F6 can be set to point to any area in RAM where the video display buffer starts. (See Table 9-1 for the dedicated addresses used to set or clear bits F0 - F6.)

The video display mode is controlled by the SAM through control bits V0 - V2. Their states must be coordinated with the VDG operating mode to obtain a valid display. (See Table 9-1 for the dedicated addresses of bits V0 through V2.)

The VDG operating mode is controlled through bits 3 - 7 of the DRB of PIA 2. The DDRB of PIA 2 is initialized by BASIC so that bits 3 - 7 of the DRB are configured for output to the VDG. The VDG operating modes are selected by writing a control code into the DRB of PIA 2 through its dedicated address. Valid settings for the video display modes and the VDG operating modes are shown in Table 9-9. As you can see, Extended Color BASIC does not take advantage of all the display modes available. The display mode column uses the abbreviation of the technical name for each display mode. The display modes indicated as NA are not supported by Extended Color BASIC.

| VDG Op Mode | SAM Control Bits | Display | BASIC Display | |
|---|---|---|---|---|
| 7 6 5 4 3 | V2 V1 V0 | Mode | PMODE | SCREEN |
| 0 x x 0 c | 0 0 0 | AI | text | 0,c |
| 0 x x 0 x | 0 0 0 | SG4 | text | 0,x |
| 0 x x 1 c | 0 0 0 | SG6 | NA | |
| 0 x x 0 x | 0 1 0 | SG8 | NA | |
| 0 x x 0 x | 1 0 0 | SG12 | NA | |
| 0 x x 0 x | 1 1 0 | SG24 | NA | |
| 1 0 0 0 c | 0 0 1 | G1C | NA | |
| 1 0 0 1 c | 0 0 1 | G1R | NA | |
| 1 0 1 0 c | 0 1 0 | G2C | NA | |
| 1 0 1 1 c | 0 1 1 | G2R | 0 | 1,c |
| 1 1 0 0 c | 1 0 0 | G3C | 1 | 1.c |
| 1 1 0 1 c | 1 0 1 | G3R | 2 | 1,c |
| 1 1 1 0 c | 1 1 0 | G6C | 3 | 1,c |
| 1 1 1 1 c | 1 1 0 | G6R | 4 | 1,c |

FF22 Set FFC5 C3 C0

x=don't care c=color set (0 or 1)

clear C4 C2 C1

Table 9-9 Available Display Modes.

To select the VDG operating mode, first use the DDRB to configure DRB bits 3 - 7 for output. Then set bit 2 of the CRB at address $FF23, enabling access to the DRB at address $FF22. Usually, BASIC has bit 2 of the CRB set. Finally, put the VDG operating mode code into the DRB at address $FF22. This whole process is shown in Listing 9-1. The VDG operating mode selected is G1R, with a color set of 1. At lines 250 - 290 the DRB is configured for output on bits 3 - 7. At lines 300 - 320, bit 2 of the CRB is set to enable access to the DRB. At lines 330 - 360, the operating mode code is put in bits 3 - 7 of the A register, and it is then stored in the DRB.

```
250         LDA $FF23        GET CRB
260         ANDA #$FB        CLEAR BIT 2
270         STA $FF23        PUT IN CRB
280         LDA #$F8         DDRB CONFG
290         STA $FF22        PUT IN DDRB
300         LDA $FF23        GET CRB
310         ORA #$04         SET BIT 2
320         STA $FF23        PUT IN CRB
330         LDA #$98         VDG=G1R, C=1
340         STA $FF22        PUT IN DRB
```

Listing 9-1 Selecting the VDG Operating Mode.

One could also rely on the way BASIC initializes the PIAs when setting a VDG operating mode. Only bits 3 - 7 of the DRB in PIA 2 are set for output and bit 2 of the CRB is set. (See Table 9-8 to verify this.) One can simply write the VDG operating mode code into the DRB. Remember that SAM control bits V2 - V0 and F6 - F0 must also be set up. Listing 9-2 is a program that selects the G2C display mode with color set zero: C = 0. The display buffer starts at address $0600. This is equivalent to BASIC's PMODE 1,0 and SCREEN 1,0 display mode.

```
100 *PROGRAM NAME: PM1
110 *SELECT PMODE 1,0 AND SCREEN 1,0. DISPLAY
120 *BUFFER STARTS AT 0600.
130 *****************************************
140 PM1      STA $FFC7       SET F0 BIT
150          STA $FFC3       SET V1 BIT
160          LDA #$A0        GET VDG MODE CODE
170          STA $FF22       SEND TO VDG
180 PMA      BRA PMA         WAIT AND OBSERVE
190 *****************************************
200          END
```

Listing 9-2 The PM1 Program.

Assemble the source code into memory with the A/IM/WE command and verify there are no errors. Go to ZBUG and run it by entering GPM1. After observing the display, press the Reset button to get back to ZBUG.

## Using The Display Modes

Each display mode is unique and only one can be in use at any one time, except AI and SG4. AI and SG4 are the same display mode with different alphanumeric and semigraphic character sets. When any display mode is selected, the contents of the buffer is displayed according to the way that display mode interprets the contents. One may also maintain several different buffer areas, displaying one at a time as RAM space permits.

The screen is divided into two areas - a border around the outside edge of the screen, and the area within the border. The outside border can be controlled least. In some display modes it is always black and in others it can be green or buff. The rectangular area within the border is the primary area controlled by the various display modes.

When using a graphics or semigraphics display mode, one determines what is displayed on the screen by controlling small areas of the screen known as **pixels**, or picture elements. A pixel is a small square or rectangular area of the screen whose color is controlled by the contents of the buffer. In a semigraphics mode one can control the color of a group of pixels. In a graphics mode, one can control the color of each pixel. How many pixels the screen is divided into depends on the display mode. Higher resolution display modes divide the screen into a larger number of smaller pixels. The pixels are arranged in rows and columns in a grid. For example, the G1R display mode divides the screen into 8192 pixels: a grid with 128 pixels across by 64 vertically.

Following are descriptions of each display mode. In the descriptions the C bit refers to VDG operating mode control bit 3, and an X indicates that what state a bit is in is of no significance.

**Alphanumeric Text (AI)**
BASIC Display Mode: SCREEN 0,C.
Border Color: Black. Buffer Size: 512

Display Format: There are 32 characters to a line and 16 lines to a full screen. The first character of the top line is determined by the first byte in the display buffer. Consecutive characters along the line, from left to right, are determined by the contents of consecutive bytes in the buffer.

Color Set: C=0: Black characters on a green background.
C=1: Black characters on an orange background.

Description: The VDG generates a dot matrix character if its video display code is in a byte in the buffer. The characters that may be displayed are: ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 !"#$%&'()*,./<>?;+:*-=\ ← ↑ []. (See Appendix C for their video display codes.)

**Semigraphics 4 (SG4)**
BASIC Display Mode: SCREEN 0,X.
Border Color: Black. Buffer Size: 512
Display Format: 64 columns, 32 rows

Display (top left corner)

| | | | | |
|---|---|---|---|---|
| $P_0$ | $P_1$ | $P_4$ | $P_5$ | . . |
| $P_2$ | $P_3$ | $P_6$ | $P_7$ | . . |
| $P_{64}$ | $P_{65}$ | | | |
| $P_{66}$ | $P_{67}$ | | | |
| . | . | | | |

Buffer Contents

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Byte 0 | 1 | $N_2$ | $N_1$ | $N_0$ | $P_0$ | $P_1$ | $P_2$ | $P_3$ |
| Byte 1 | 1 | $N_2$ | $N_1$ | $N_0$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ |
| . | . | | | | | | | |
| . | . | | | | | | | |
| Byte 32 | 1 | $N_2$ | $N_1$ | $N_0$ | $P_{64}$ | $P_{65}$ | $P_{66}$ | $P_{67}$ |
| . | . | | | | | | | |

Color Set:

| $P_x$ | $N_2$ | $N_1$ | $N_0$ | |
|---|---|---|---|---|
| 0 | x | x | x | - Black |
| 1 | 0 | 0 | 0 | - Green |
| 1 | 0 | 0 | 1 | - Yellow |
| 1 | 0 | 1 | 0 | - Blue |
| 1 | 0 | 1 | 1 | - Red |
| 1 | 1 | 0 | 0 | - Buff |
| 1 | 1 | 0 | 1 | - Cyan |
| 1 | 1 | 1 | 0 | - Magenta |
| 1 | 1 | 1 | 1 | - Orange |

Description: The screen is divided into 2048 pixels; each group of four pixels is controlled by one byte in the buffer. This is actually the AI display mode, but the characters displayed are semigraphic. Each group of pixels ($P_0$ - $P_3$ e.g.) corresponds to the position of a displayable text character. Three bits ($N_2$, $N_1$, and $N_0$) control the color of all the pixels in a group, and four bits ($P_0$, $P_1$, $P_2$, and $P_3$) are used to turn on (bit set) or off (bit clear) each pixel within the group. When a pixel is off it is black; otherwise it is the assigned color of that group. The MSB of a byte in the buffer must be set for a pixel group to be controlled as described.

**Semigraphics 6 (SG6)**
BASIC Display Mode: Not supported.
Border Color: Black. Buffer Size: 512
Display Format: 64 columns, 48 rows

Display (top left corner)

| | | | | |
|---|---|---|---|---|
| $P_0$ | $P_1$ | $P_6$ | $P_7$ | . . |
| $P_2$ | $P_3$ | $P_8$ | $P_9$ | . . |
| $P_4$ | $P_5$ | $P_{10}$ | $P_{11}$ | |
| $P_{198}$ | $P_{199}$ | | | |
| $P_{200}$ | $P_{201}$ | | | |
| $P_{202}$ | $P_{203}$ | | | |
| . | . | | | |

Buffer Contents

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Byte 0 | 1 | N | $P_0$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
| Byte 1 | 1 | N | $P_6$ | $P_7$ | $P_8$ | $P_9$ | $P_{10}$ | $P_{11}$ |
| . | . | | | | | | | |
| . | . | | | | | | | |
| . | . | | | | | | | |
| Byte 32 | 1 | N | $P_{198}$ | $P_{199}$ | $P_{200}$ | $P_{201}$ | $P_{202}$ | $P_{203}$ |
| . | . | | | | | | | |

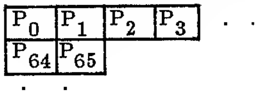
Color Set:

| C=0 N | $P_x$ | | C=1 N | $P_x$ | |
|---|---|---|---|---|---|
| x | 0 | - Black | x | 0 | - Black |
| 0 | 1 | - Blue | 0 | 1 | - Magenta |
| 1 | 1 | - Red | 1 | 1 | - Orange |

Description: The screen is divided into 3072 pixels, and each group of six pixels ($P_0$ - $P_5$ e.g.) is controlled by one byte. The N bit controls the color of all pixels in a group, and bits $P_0$ - $P_5$ turn on (bit set) or off (bit clear) each pixel. When a pixel is off it is black; otherwise it is the assigned color of that group. There are two color sets determined by VDG operating mode control bit 3 (C). The MSB of a byte in the buffer must be set for a pixel group to be controlled as described.

**Semigrapghics 8 (SG8)**
BASIC Display Mode: Not supported.
Border Color: Black. Buffer Size: 2048
Display Format: 64 columns, 64 rows

Display (top left corner)



Buffer Contents

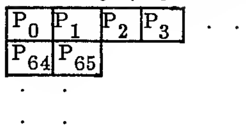
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Byte 0 | 1 | $N_2$ | $N_1$ | $N_0$ | $P_0$ | $P_1$ | X | X |
| Byte 1 | 1 | $N_2$ | $N_1$ | $N_0$ | $P_2$ | $P_3$ | X | X |
| . | | | | . | | | | |
| . | | | | . | | | | |
| Byte 32 | 1 | $N_2$ | $N_1$ | $N_0$ | $P_{64}$ | $P_{65}$ | X | X |
| . | | | | . | | | | |

| Color Set: | $P_x$ | $N_2$ | $N_1$ | $N_0$ | |
|---|---|---|---|---|---|
| | 0 | x | x | x | - Black |
| | 1 | 0 | 0 | 0 | - Green |
| | 1 | 0 | 0 | 1 | - Yellow |
| | 1 | 0 | 1 | 0 | - Blue |
| | 1 | 0 | 1 | 1 | - Red |
| | 1 | 1 | 0 | 0 | - Buff |
| | 1 | 1 | 0 | 1 | - Cyan |
| | 1 | 1 | 1 | 0 | - Magenta |
| | 1 | 1 | 1 | 1 | - Orange |

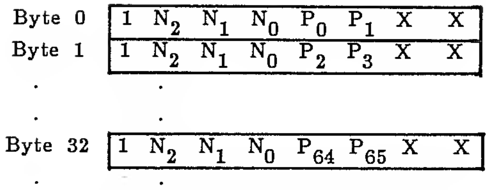
Description: The screen is divided into 4096 pixels; each group of two pixels ($P_0$ - $P_1$ e.g.) is controlled by one byte. Bits $N_2$ - $N_0$ control the color of all the pixels in a group. Bits $P_0$ and $P_1$ turn on (bit set) or off (bit clear) their corresponding pixels within that group. When a pixel is off it is black; otherwise it is the assigned color of that group. The MSB of a byte in the buffer must be set for a pixel group to be controlled as described.

**Semigraphics 12 (SG12)**
BASIC Display Mode: Not supported.
Border Color: Black. Buffer Size: 3072
Display Format: 64 columns, 96rows

Display (top left corner)



Buffer Contents


Byte 0 1 $N_2$ $N_1$ $N_0$ $P_0$ $P_1$ X X
Byte 1 1 $N_2$ $N_1$ $N_0$ $P_2$ $P_3$ X X
Byte 32 1 $N_2$ $N_1$ $N_0$ $P_{64}$ $P_{65}$ X X

Color Set:

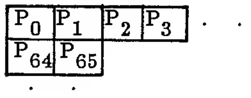
| $P_x$ | $N_2$ | $N_1$ | $N_0$ | |
|---|---|---|---|---|
| 0 | x | x | x | - Black |
| 1 | 0 | 0 | 0 | - Green |
| 1 | 0 | 0 | 1 | - Yellow |
| 1 | 0 | 1 | 0 | - Blue |
| 1 | 0 | 1 | 1 | - Red |
| 1 | 1 | 0 | 0 | - Buff |
| 1 | 1 | 0 | 1 | - Cyan |
| 1 | 1 | 1 | 0 | - Magenta |
| 1 | 1 | 1 | 1 | - Orange |

Description: The screen is divided into 6144 pixels, each group of two pixels ($P_0$ - $P_1$ e.g.) is controlled by one byte. Bits $N_2$ - $N_0$ control the color of all the pixels in a group. Bits $P_0$ and $P_1$ turn on (bit set) or off (bit clear) their corresponding pixels within that group. When a pixel is off it is black; otherwise it is the assigned color of that group. The MSB of a byte in the buffer must be set for a pixel group to be controlled as described.

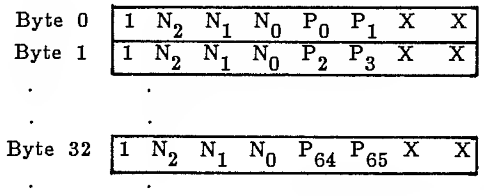
**Semigraphics 24 (SG24)**
BASIC Display Mode: Not supported.
Border Color: Black.
Display Format: 64 columns, 192 rows

Buffer Size: 6144

Display (top left corner)



Buffer Contents


Byte 0 1 $N_2$ $N_1$ $N_0$ $P_0$ $P_1$ X X
Byte 1 1 $N_2$ $N_1$ $N_0$ $P_2$ $P_3$ X X
Byte 32 1 $N_2$ $N_1$ $N_0$ $P_{64}$ $P_{65}$ X X

Color Set:

| $P_x$ | $N_2$ | $N_1$ | $N_0$ | |
|---|---|---|---|---|
| 0 | x | x | x | - Black |
| 1 | 0 | 0 | 0 | - Green |
| 1 | 0 | 0 | 1 | - Yellow |
| 1 | 0 | 1 | 0 | - Blue |
| 1 | 0 | 1 | 1 | - Red |
| 1 | 1 | 0 | 0 | - Buff |
| 1 | 1 | 0 | 1 | - Cyan |
| 1 | 1 | 1 | 0 | - Magenta |
| 1 | 1 | 1 | 1 | - Orange |

Description: The screen is divided into 12288 pixels; each group of two pixels ($P_0$ - $P_1$ e.g.) is controlled by one byte. Bits $N_2$ - $N_0$ control the color of all the pixels in a group. Bits $P_0$ and $P_1$ turn on (bit set) or off (bit clear) their corresponding pixels within that group. When a pixel is off it is black; otherwise it is the assigned color of that group. The MSB of each byte in the buffer must be set for a pixel group to be controlled as described.

**Graphics 1 with Color (G1C)**

BASIC Display Mode: Not supported.

Border Color: If C=0, then green.
If C=1, then buff.

Buffer Size: 1024

Display Format: 64 columns, 64 rows

Display (top left corner)

| | | | | |
|---|---|---|---|---|
| $P_0$ | $P_1$ | $P_2$ | $P_3$ | . . |
| $P_{64}$ | $P_{65}$ | $P_{66}$ | $P_{67}$ | |
| . . | | | | |
| . . | | | | |

Buffer Contents

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Byte 0 | $N_{10}$ | $N_{00}$ | $N_{11}$ | $N_{01}$ | $N_{12}$ | $N_{02}$ | $N_{13}$ | $N_{03}$ |
| . | | . | | | | | | |
| . | | . | | | | | | |
| Byte 16 | $N_{164}$ | $N_{064}$ | $N_{165}$ | $N_{065}$ | $N_{166}$ | $N_{066}$ | $N_{167}$ | $N_{067}$ |
| . | | . | | | | | | |

Color Set:

| C=0 $N_1$ | $N_0$ | | C=1 $N_1$ | $N_0$ | |
|---|---|---|---|---|---|
| 0 | 0 | - Green | 0 | 0 | - Buff |
| 0 | 1 | - Yellow | 0 | 1 | - Cyan |
| 1 | 0 | - Blue | 1 | 0 | - Magenta |
| 1 | 1 | - Red | 1 | 1 | - Orange |

Description: The screen is divided into 4096 pixels; each is controlled by two bits of a byte in the buffer. One byte controls four pixels. The two bits $N_{1x}$ and $N_{0x}$ determine the color of their pixel, $P_x$. All pixels are always on: not black. Only their color can be changed.

**Graphics 1 with Resolution (G1R)**

BASIC Display Mode: Not supported.

Border Color: If C=0, then green.
If C=1, then buff.

Buffer Size: 1024

Display Format: 128 columns, 64 rows

Display (top left corner)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $P_0$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ | . . |
| $P_{128}$ | $P_{129}$ | $P_{130}$ | $P_{131}$ | $P_{132}$ | $P_{133}$ | $P_{134}$ | $P_{135}$ | . . |
| . . | | | | | | | | |
| . . | | | | | | | | |

Buffer Contents

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Byte 0 | $P_0$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ |
| . . | . | | . | | | | | |
| Byte 16 | $P_{128}$ | $P_{129}$ | $P_{130}$ | $P_{131}$ | $P_{132}$ | $P_{133}$ | $P_{134}$ | $P_{135}$ |
| . . | . | | . | | | | | |

Color Set:

| C | $P_x$ | |
|---|---|---|
| 0 | 0 | - Black |
| 0 | 1 | - Green |
| 1 | 0 | - Black |
| 1 | 1 | - Buff |

Description: The screen is divided into 8192 pixels; each is controlled by one bit of a byte in the buffer. Each byte controls eight pixels. When a bit is set, its corresponding pixel is turned on and has a color determined by the C bit. When a bit is clear, the pixel is turned off and is black. Bit $P_x$ corresponds to pixel $P_x$.

**Graphics 2 with Color (G2C)**
BASIC Display Mode: Not supported.
Border Color: If C=0, then green. Buffer Size: 2048
If C=1, then black.
Display Format: 128 columns, 64 rows

Display (top left corner)

| | | | | |
|---|---|---|---|---|
| $P_0$ | $P_1$ | $P_2$ | $P_3$ | . . |
| $P_{128}$ | $P_{129}$ | $P_{130}$ | $P_{131}$ | . . |
| . . | | | | |
| . . | | | | |

Buffer Contents

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Byte 0 | $N_{10}$ | $N_{00}$ | $N_{11}$ | $N_{01}$ | $N_{12}$ | $N_{02}$ | $N_{13}$ | $N_{03}$ |
| . | | . | | | | | | |
| . | | . | | | | | | |
| Byte 32 | $N_{1128}$ | $N_{0128}$ | $N_{1129}$ | $N_{0129}$ | $N_{1130}$ | $N_{0130}$ | $N_{1131}$ | $N_{0131}$ |
| . | | . | | | | | | |

Color Set:

| C=0 $N_1$ | $N_0$ | | C=1 $N_1$ | $N_0$ | |
|---|---|---|---|---|---|
| 0 | 0 | Green | 0 | 0 | Buff |
| 0 | 1 | Yellow | 0 | 1 | Cyan |
| 1 | 0 | Blue | 1 | 0 | Magenta |
| 1 | 1 | Red | 1 | 1 | Orange |

Description: The screen is divided into 8192 pixels; each is controlled by two bits of a byte in the buffer. One byte controls four pixels. The bits $N_{1x}$ and $N_{0x}$ determine the color of their pixel, $P_x$. All pixels are always on: not black. Only their color can be changed.

**Graphics 2 with Resolution (G2R)**
BASIC Display Mode: PMODE 0: SCREEN 1,C
Border Color: If C=0, then green. Buffer Size: 1536
If C=1, then buff.
Display Format: 128 columns, 96 rows

Display (top left corner)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $P_0$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ | . . |
| $P_{128}$ | $P_{129}$ | $P_{130}$ | $P_{131}$ | $P_{132}$ | $P_{133}$ | $P_{134}$ | $P_{135}$ | . . |
| . . | | | | | | | | |
| . . | | | | | | | | |

Buffer Contents

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Byte 0 | $P_0$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ |
| . . | | . | | | . | | | |
| Byte 16 | $P_{128}$ | $P_{129}$ | $P_{130}$ | $P_{131}$ | $P_{132}$ | $P_{133}$ | $P_{134}$ | $P_{135}$ |
| . . | | . | | | . | | | |

Color Set:

| C | $P_x$ | |
|---|---|---|
| 0 | 0 | Black |
| 0 | 1 | Green |
| 1 | 0 | Black |
| 1 | 1 | Buff |

Description: The screen is divided into 12288 pixels; each is controlled by one bit of a byte in the buffer. Each byte controls eight pixels. When a bit is set, its corresponding pixel is turned on and has a color determined by the C bit. When a bit is clear, the pixel is turned off and is black. Bit $P_x$ corresponds to pixel $P_x$.

**Graphics 3 with Color (G3C)**
BASIC Display Mode: PMODE 1: SCREEN 1,C
Border Color: If C=0, then green.
If C=1, then buff.
Buffer Size: 3072
Display Format: 128 columns, 96 rows

Display (top left corner)

| | | | | |
|---|---|---|---|---|
| $P_0$ | $P_1$ | $P_2$ | $P_3$ | . . |
| $P_{128}$ | $P_{129}$ | $P_{130}$ | $P_{131}$ | . . |
| . . | | | | |
| . . | | | | |

Buffer Contents

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Byte 0 | $N_{10}$ | $N_{00}$ | $N_{11}$ | $N_{01}$ | $N_{12}$ | $N_{02}$ | $N_{13}$ | $N_{03}$ |
| . | | . | | | | | | |
| . | | . | | | | | | |
| Byte 32 | $N_{1128}$ | $N_{0128}$ | $N_{1129}$ | $N_{0129}$ | $N_{1130}$ | $N_{0130}$ | $N_{1131}$ | $N_{0131}$ |
| . | | . | | | | | | |

| Color Set: | $N_1$ | $N_0$ | | | $N_1$ | $N_0$ | |
|---|---|---|---|---|---|---|---|
| C=0 | 0 | 0 | - Green | C=1 | 0 | 0 | - Buff |
| | 0 | 1 | - Yellow | | 0 | 1 | - Cyan |
| | 1 | 0 | - Blue | | 1 | 0 | - Magenta |
| | 1 | 1 | - Red | | 1 | 1 | - Orange |

Description: The screen is divided into 12288 pixels; each is controlled by two bits of a byte in the buffer. One byte controls four pixels. The two bits $N_{1x}$ and $N_{0x}$ determine the color of their pixel, $P_x$. All the pixels are always on: not black. Only their color can be changed.

**Graphics 3 with Resolution (G3R)**
BASIC Display Mode: PMODE 2: SCREEN 1,C
Border Color: If C=0, then green.
If C=1, then buff.
Buffer Size: 3072
Display Format: 128 columns, 192 rows

Display (top left corner)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $P_0$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ | . . |
| $P_{192}$ | $P_{193}$ | $P_{194}$ | $P_{195}$ | $P_{196}$ | $P_{197}$ | $P_{198}$ | $P_{199}$ | . . |
| . . | | | | | | | | |
| . . | | | | | | | | |

Buffer Contents

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Byte 0 | $P_0$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ |
| . . | . | | | . | | | | |
| Byte 32 | $P_{192}$ | $P_{193}$ | $P_{194}$ | $P_{195}$ | $P_{196}$ | $P_{197}$ | $P_{198}$ | $P_{199}$ |
| . . | . | | | . | | | | |

Color Set:

| C | $P_x$ | |
|---|---|---|
| 0 | 0 | - Black |
| 0 | 1 | - Green |
| 1 | 0 | - Black |
| 1 | 1 | - Buff |

Description: The screen is divided into 24576 pixels; each is controlled by one bit of a byte in the buffer. Each byte controls eight pixels. When a bit is set, its corresponding pixel is turned on and has a color determined by the C bit. When a bit is clear, the pixel is turned off and is black. Bit $P_x$ corresponds to pixel $P_x$.

**Graphics 6 with Color (G6C)**
BASIC Display Mode: PMODE 3: SCREEN 1,C
Border Color: If C=0, then green.
If C=1, then buff.
Buffer Size: 6144
Display Format: 128 columns, 192 rows

Display (top left corner)

| | | | | |
|---|---|---|---|---|
| $P_0$ | $P_1$ | $P_2$ | $P_3$ | . . |
| $P_{192}$ | $P_{193}$ | $P_{194}$ | $P_{195}$ | . . . |
| . . | | | | |
| . . | | | | |

Buffer Contents

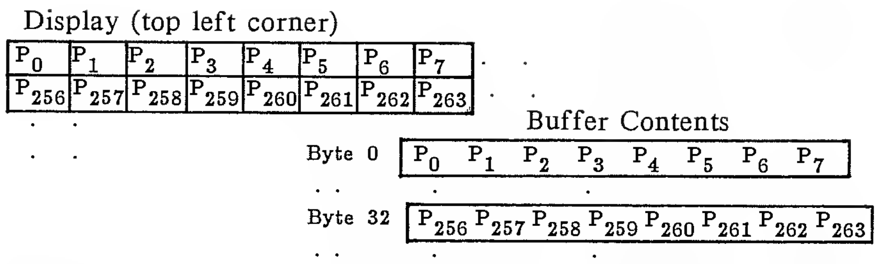
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Byte 0 | $N_{10}$ | $N_{00}$ | $N_{11}$ | $N_{01}$ | $N_{12}$ | $N_{02}$ | $N_{13}$ | $N_{03}$ |
| . | | | . | | | | | |
| . | | | . | | | | | |
| Byte 32 | $N_{1192}$ | $N_{0192}$ | $N_{1193}$ | $N_{0193}$ | $N_{1194}$ | $N_{0194}$ | $N_{1195}$ | $N_{0195}$ |
| . | | | . | | | | | |

Color Set:

| C=0 $N_1$ | $N_0$ | | C=1 $N_1$ | $N_0$ | |
|---|---|---|---|---|---|
| 0 | 0 | - Green | 0 | 0 | - Buff |
| 0 | 1 | - Yellow | 0 | 1 | - Cyan |
| 1 | 0 | - Blue | 1 | 0 | - Magenta |
| 1 | 1 | - Red | 1 | 1 | - Orange |

Description: The screen is divided into 24576 pixels; each is controlled by two bits of a byte in the buffer. One byte controls four pixels. The two bits $N_{1x}$ and $N_{0x}$ determine the color of their pixel, $P_x$. All the pixels are always on: not black. Only their color can be changed.

**Graphics 6 with Resolution (G6R)**
BASIC Diaplay Mode: PMODE 4: SCREEN 1,C
Border Color: If C=0, then green.
If C=1, then buff.
Buffer Size: 6144
Display Format: 256 columns, 192 rows


Display (top left corner)
$P_0$ $P_1$ $P_2$ $P_3$ $P_4$ $P_5$ $P_6$ $P_7$
$P_{256}$ $P_{257}$ $P_{258}$ $P_{259}$ $P_{260}$ $P_{261}$ $P_{262}$ $P_{263}$
Buffer Contents
Byte 0
$P_0$ $P_1$ $P_2$ $P_3$ $P_4$ $P_5$ $P_6$ $P_7$
Byte 32
$P_{256}$ $P_{257}$ $P_{258}$ $P_{259}$ $P_{260}$ $P_{261}$ $P_{262}$ $P_{263}$

Color Set:

| C | $P_X$ | |
|---|---|---|
| 0 | 0 | - Black |
| 0 | 1 | - Green |
| 1 | 0 | - Black |
| 1 | 1 | - Buff |

Description: The screen is divided into 49152 pixels; each is controlled by one bit of a byte in the buffer. Each byte controls eight pixels. When a bit is set, its corresponding pixel is turned on and has a color determined by the C bit. When a bit is clear, the pixel is turned off and is black. Bit $P_X$ corresponds to pixel $P_X$.

**A Graphics Display Exercise**

Listing 9-3 is an example of using the G6R graphics display mode in an assembly language program. The items that must be set up are: the SAM control bits, F6 - F0 and V2 - V0, and the VDG operating mode. The program establishes the video buffer starting at address $1000. All bytes within the buffer are cleared, resulting in a black screen. Then a green rectangular outline is drawn in the upper left corner of the screen.

```
100 *PROGRAM NAME: GRPH
110 *THIS PROGRAM WILL SET UP DISPLAY MODE G6R.
120 *THE VIDEO DISPLAY BUFFER WILL START AT HEX
130 *ADDRESS 1000. THE SCREEN IS SET TO ALL BLACK
140 *AND A RECTANGLE IS DRAWN IN THE UPPER LEFT
150 *CORNER.
160 ***********************************************
170 GRPH     STA $FFC8        CLEAR F1
180          STA $FFCD        SET F3
190          STA $FFC5        SET V2
200          STA $FFC3        SET V1
210          LDA #$F0         VDG OP MODE
220          STA $FF22        SET VDG OP MODE
230          CLRA             GET 00
240          LDX #$1000       START OF BUFFER
250 GRA      STA ,X+          CLEAR BUFFER
260          CMPX #$2800      END OF BUFFER?
270          BLO GRA          IF NOT,DO AGAIN
280          LDX #$1104       START OF TOP LINE
290          LDA #$FF         BUFFER CODE
300 GRB      STA ,X+          PUT IN BUFFER
310          CMPX #$1110      END OF LINE?
320          BLS GRB          IF NOT,DO AGAIN
```

```
330         LDX #$1604      START OF BOT LINE
340 GRC     STA ,X+         PUT IN BUFFER
350         CMPX #$1610     END OF LINE?
360         BLS GRC         IF NOT,DO AGAIN
370         LDA #$80        LEFT LINE CODE
380         LDX #$1124      START OF LEFT LINE
390 GRD     STA ,X          PUT IN BUFFER
400         LEAX 32,X       GOTO NEXT LOWER PIXEL
410         CMPX #$1604     END OF LINE?
420         BLO GRD         IF NOT,DO AGAIN
430         LDA #$01        RIGHT LINE CODE
440         LDX #$1130      START OF RIGHT LINE
450 GRE     STA ,X          PUT IN BUFFER
460         LEAX 32,X       NEXT LOWER PIXEL
470         CMPX #$1610     END OF LINE?
480         BLO GRE         IF NOT,DO AGAIN
490 GREND   BRA GREND       WAIT AND OBSERVE
500 *********************************************
510         END
```

Listing 9-3 The GRPH Program.

In the source code lines 170 and 180 set the starting address of the video buffer area to $1000. Lines 190 and 200 set the proper SAM video control mode for the G6R display mode. Lines 210 and 220 set the VDG operating mode. Lines 230 - 270 clear the buffer area, resulting in a black screen. Lines 280 - 480 draw the four sides of the rectangle. At line 490 the program is in a loop that lets the operator view the display.

Assemble the source code in memory with the A/IM/WE command and verify there are no errors. Go to ZBUG and run it by entering: GGRPH. To get back to ZBUG after viewing the display press the Reset button.

## INTERRUPTS

In this section are described the interrupt sources and how to control their operation. The IRQ interrupt originates from devices in the Color Computer, and the NMI and FIRQ from outside. IRQ and FIRQ are sent to the MPU from the PIAs. It is imperative that one knows how to use the PIAs to control the IRQ and FIRQ interrupts.

### Interrupt Sources

The IRQ and FIRQ interrupts are sent to the MPU from PIAs 1 and 2. The IRQ interrupt sources are internal to the Color Computer, and are connected to CA1 and CB1 of PIA 1. The FIRQ interrupt sources are external to the computer and are connected to CA1 and CB1 of PIA 2. The NMI interrupt signal line is connected to the ROM cartridge slot. Fig. 9-6 is a simplified block diagram showing the connections between the PIAs, MPU, and the interrupt sources of the Color Computer. One can see there are a total of five signals which can be used to generate an interrupt. Two signals can generate an IRQ interrupt, two can generate an FIRQ interrupt, and one generates an NMI interrupt.

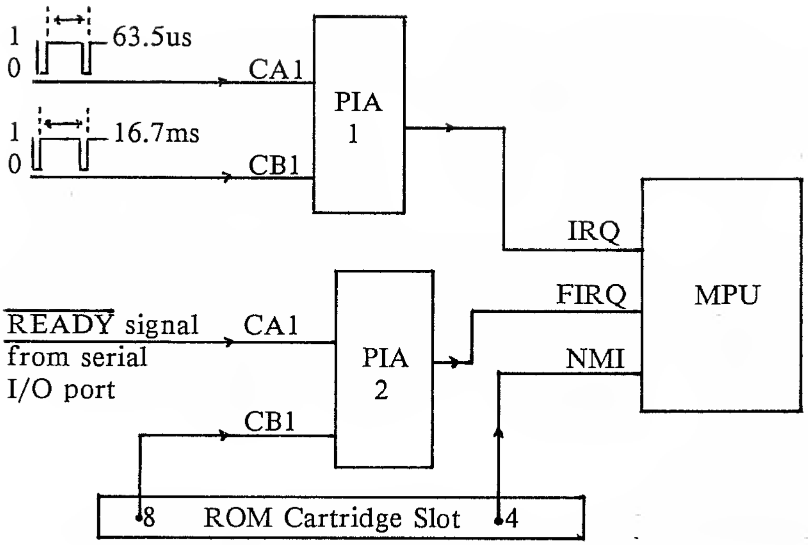


Fig. 9-6 Block Diagram of the Interrupt Sources.

■ IRQ Interrupts - PIA 1 is connected to the IRQ interrupt signal going to the MPU. At CA1, an electrical signal from the VDG creates a transition from a high to a low (or from a low to a high) once every 63.5 microseconds, or 15,478 times a second. This signal is always present and can be used to generate an IRQ interrupt at its repetition rate. At CB1 is an electrical signal from the VDG that creates a transition from a high to a low (or from a low to a high) once every 16.7 milliseconds or 60 times a second. The signal at CB1 is always present and can also be used to generate an IRQ interrupt at its repetition rate.

It appears that the signal at CA1 would cause an interrupt too often to be of much use. The MPU requires about 20 microseconds to perform the IRQ interrupt sequence and another 15 microseconds for the RTI instruction at the end of the interrupt handler. This leaves about 28 microseconds for the interrupt handler to perform some task before the interrupt occurs again. This is not enough time to perform very much; about 4 to 10 instructions. The signal at CB1, when the PIA is initialized, causes an interrupt at the fixed rate of 60 times a second, a much more useful rate.

■ FIRQ Interrupts - PIA 2 is connected to the FIRQ interrupt signal going to the MPU. The $\overline{\text{READY}}$ signal from the serial I/O port is connected to CA1. A device connected to the serial I/O port, when it goes from a non-ready state to a ready state, causes the $\overline{\text{READY}}$ signal to go from a high to a low. If PIA 2 is configured to sense a high-to-low transition at CA1, an FIRQ interrupt will be generated when the device becomes ready. However, not all devices send a READY signal to the computer. Normally, this signal means the device is turned on, not busy doing something, and is ready to transfer data to or receive data from the computer. If PIA 2 is configured to sense a low-to-high transition at CA1, an FIRQ interrupt will be generated when the device becomes non-ready.

The CB1 pin of PIA 2 is connected to pin 8 of the ROM cartridge connector. A cartridge plugged in may cause an FIRQ interrupt if CB1 is configured to sense it. The ROM cartridge slot is described in Chapter 10.

■ NMI Interrupts - The NMI interrupt signal line to the MPU is connected directly to pin 4 of the ROM cartridge slot. Thus, a cartridge may directly cause an NMI interrupt. Note that an NMI interrupt does not require any PIAs to be configured, and that it is not maskable. The interrupt sequence is initiated by a temporary low signal on pin 4 of the ROM cartridge slot connector. The ROM cartridge slot is described in Chapter 10.

**Controlling PIA Interrupts**

A PIA must be initialized to send an interrupt to the MPU. This is done by setting or clearing bits in the control registers, CRA and/or CRB, through their dedicated addresses. They are bits 0 and 1 of control register A or B, which control interrupts sensed by CA1 or CB1. Table 9-10 shows the dedicated addresses to use to access the control registers to control the IRQ and FIRQ interrupts.

| Int. Type | Signal | Dedicated Address |
|---|---|---|
| IRQ | CA1 | FF01 |
| | CB1 | FF03 |
| FIRQ | CA1 | FF21 |
| | CB1 | FF23 |

Table 9-10 Dedicated Addresses of PIA Control Registers.

If bit 0 of CRA(B) is clear, that half of a PIA will not generate an interrupt when the CA(B)1 input is activated. If bit 0 is set, that half of a PIA will generate an interrupt when the CA(B)1 input is activated. In either case, bit 7 of CRA(B) is set when the CA(B)1 input is activated.

If bit 1 of CRA(B) is clear, the CA(B)1 input will be activated when it sees a high-to-low transition. If bit 1 of CRA(B) is set, that input will be activated upon seeing a low-to-high transition.

After a half of a PIA (A or B) has sensed an activating transition at CA(B)1, bit 7, the flag bit of the CRA(B) will be set, and an interrupt may be sent to the MPU depending on the state of bit 0 of the respective control register. Then the flag bit must be cleared for that half of the PIA to be able to sense another CA(B)1 activation. Clearing the flag bit also causes the PIA to stop sending an active interrupt signal to the MPU if bit 0 of the control register is set. If this is not done, the MPU will be interrupted again as soon as the F or I bit of the CC register is cleared. The flag bit is cleared by having the MPU read from the corresponding data register (DRA or DRB).

**Programming Responsibilities**

A program using interrupts must initialize the PIA control register (either CRA or CRB) for the MPU to be able to receive an IRQ or FIRQ interrupt. This entails setting or clearing bit 1 and setting bit 0 of a control register. However, take care not to modify the other bits in the control register if they are being used to control another condition. The progam must also ensure the flag bit is clear, by reading from the data register of the half of the PIA to be used to generate interrupts. Listing 9-4 provides instructions that will set bit 0 of the CRB of PIA 1 and ensure the flag bit is clear, allowing IRQ interrupts to be sensed at CB1 at the rate of 60 per second.

```
LDA $FF03        READ CRB
ORA #$05         SET BIT 0+2
STA $FF03        STORE IN CRB
LDA $FF02        READ DRB(CLR FLAG BIT)
```

Listing 9-4 Enabling the IRQ Interrupt.

The LDA reads the contents of the CRB and the ORA sets bits 0 and 2. Bit 2 set allows access to the data register. STA stores the result back in the CRB. The last LDA reads the contents of DRB, clearing the flag bit.

An interrupt handler has the responsibility of clearing the corresponding flag bit in the control register before returning to the interrupted program. Again, this is done by reading the contents of the corresponding data register. An example of a CB1-PIA 1 interrupt handler performing this function and returning is shown below:

```
LDB $FF02        READ DRB(CLR FLAG BIT)
RTI              RETURN
```

This procedure was performed in the SAMPL1 program at the end of Chapter 7. At lines 320 - 340 the PIA is initialized, and at lines 1630 and 1640 the interrupt handler clears the flag bit and returns.

The program responsibilities required for using interrupts from a PIA are in addition to other program duties, such as establishing an S stack, setting up the vector jump instructions, and enabling or disabling, via the I and F bits of the CC register, the MPU's reception of the interrupt signals. All the duties involved in using interrupts from CB1-PIA 1 are shown in a skeleton program in Fig. 9-7. To use interrupts from the other side of PIA 1 or from PIA 2, the dedicated address must be changed to access the corresponding PIA registers.

```
         ORG $2000
         ORCC #$50       SET I AND F BITS
         LDS #30+NSTK    ESTABLISH STACK
         LDA #$7E        JUMP OP CODE
         STA $010C       PUT IN VECT TABL
         LDX #INHDL      ADDR OF INT HANDLER
         STX $010D       PUT IN VECT TABL
          - - - -
          - - - -
          - - - -
         LDA $FF03       READ CRB
         ORA #$05        SET BITS 0+2
         STA $FF03       PUT IN CRB
         LDA $FF02       READ DRB(CLR FLAG)
         ANDCC #$EF      CLEAR I BIT
          - - - -
          - - - -
          - - - -
NSTK     RMB 30          STACK AREA
```

```
INHDLR    - - - -        INT. HANDLER
          - - - -
          - - - -
          - - - -
         LDA $FF02       READ DRB(CLR FLAG)
         RTI             RETURN
```

Fig. 9-7 A Skeleton Program Using IRQ Interrupts.

In contrast, the NMI interrupt comes directly to the MPU from the ROM cartridge slot and not through a PIA. Thus, the programming responsibilities exclude all references to any PIAs unless there is a PIA in the cartridge.

# CHAPTER 10

# Technical Details

This chapter will describe the rest of the Color Computer's internal devices and cartridge connector. The internal devices are used to input and/or output data to cassette, any device connected to the serial I/O port, or joysticks, and to generate sound. Knowing how to control and use the internal devices lets one write subroutines to input or output data or perform functions with non-standard peripheral equipment connected to the computer.

The cartridge connector provides access to the main buses: the data bus, address bus, part of the control bus, and various other signals. A cartridge plugged into the connector can add more ROM, RAM, PIAs, or any other desired components. An example is the disk units, which are connected via the cartridge connector. A cartridge could also contain a sound synthesizer to increase sound-making capabilities, or a speech synthesizer to provide voice.

## INTERNAL DEVICES

All the internal devices are controlled through the PIAs. Any device that can be used or controlled is connected by wiring to PIA signals: the data path bits PA(B)0 - 7, CA(B)1, and/or CA(B)2 signals. A PIA that sends data or a control code to a device must have those bits configured for output. A PIA that receives data or signals from a device must have those bits configured for input. A data register's bits are configured with the respective data direction register. A PIA that sends a CA2 or CB2 signal to a device must have configured that signal to output by setting up bits 3, 4, and 5 of the respective control register. No devices in the Color Computer send signals to any CA2 or CB2. Any PIA that receives a signal at CA1 or CB1 must have configured them to detect the desired transition and have enabled or disabled their interrupt by setting up bits 0 and 1 of the respective control register. No devices receive signals from CA1 and CB1 of any PIA, thus PIAs can only input from CA1 and CB1. This will all become evident later as the diagrams of the PIAs and other devices are presented and described.

The connections between the PIAs and the other devices are shown in simplified block diagrams in Figs. 10-1, 10-2, and 10-3. Each data line or signal line has an arrow on it to indicate the normal flow of data. The connectors (jacks) shown in the diagrams are drawn as they appear when viewed from the rear of the computer, except for the cartridge connector which is drawn as viewed from the right side. One may purchase Radio Shack's **TRS-80 Color Computer Technical Reference Manual**, catalog number 26-3193 (or the Color Computer 2 service manual), for more detailed electronic schematics.

In the diagrams and descriptions, signals are described as a voltage. A volt is a unit of measurement of electric potential, in the same way a foot is a unit of measurement of distance. Both measure a difference between two points. In the descriptions of signals, a signal's voltage is defined as the voltage difference between that signal and the computer's frame or chassis. An electrical connection to the chassis, or ground as it is also called, is represented by the ⏚ symbol. A high signal has a voltage of four to five volts. A low signal has a voltage of approximately zero volts.

**The Keyboard**

The keyboard is composed of 52 switches, with one switch under each key. The wiring to and from the switches is arranged as a grid of columns and rows, shown in Fig. 10-1. Data paths A and B of PIA 1, except PA7, are used to determine which key, if any, is currently being depressed. When a key is depressed the switch beneath it is turned on, causing a connection between a PBx output and a PAx input.

The keyboard is interrogated by sending out a low (a zero bit) on only one B side data line, and sending out highs (one bits) on all the other B side data lines. Then the A side data register of PIA 1 is read. Reading any zero bit except PA7 indicates that a key has been depressed in the column with a low signal. This process is repeated by sequentially sending out a low to each column and reading the DRA to see if any or which key is depressed. The ROM POLCAT subroutine does this and calculates the ASCII code of the depressed key.

PA7 is not connected to the keyboard. Writing $F7 into the DRB will set data line PB3 low and all the others high. If the S key is depressed, the MPU would read a binary value of x1111011, or DRA2 clear, from the DRA. DRA7 may be in any state. If the DRA is read when no key is depressed, the binary value read will be x1111111. This is because the PA0 - PA6 inputs sense a high if nothing is connected to them; this being the case when no key is depressed.

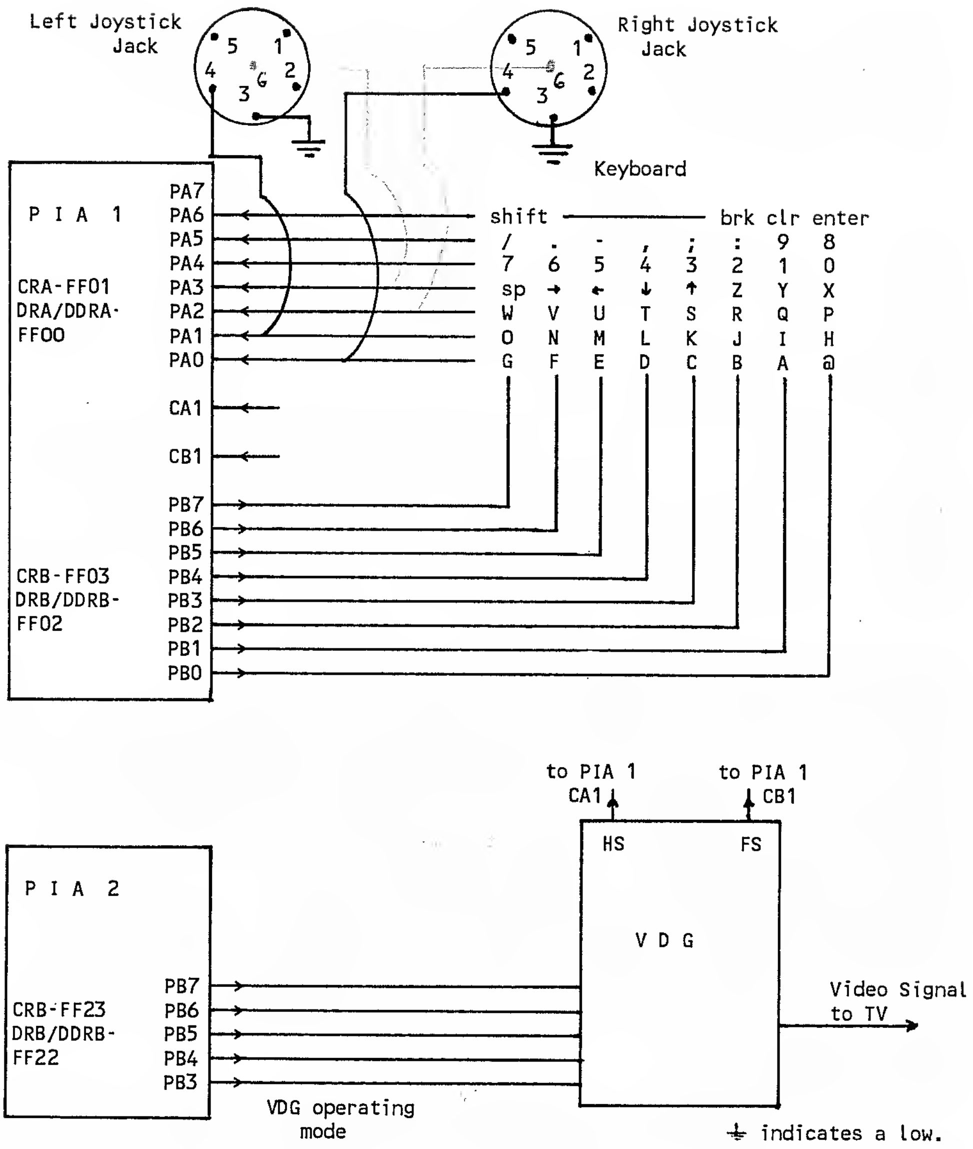


Figure 10-1
Keyboard, VDG, and Joystick Fire Button Diagram.

**Joystick Fire Buttons**

Each joystick has a fire button which actuates a switch. When the button is depressed, the switch is closed, making an electrical connection to a low. In Fig. 10-1, the left joystick fire button is connected to PA1 of PIA 1, and the right to PA0. When a fire button is not depressed the corresponding data line is high. While a fire button is depressed the corresponding data line, PA0 or PA1, is low. Thus, one can simply read the DRA of PIA 1 to see if either button is depressed. Bit 0 clear indicates the right fire button is depressed and bit 1 clear indicates the left button is depressed. The data lines, PA1 and PA0, are shown connected to pin 4 of their respective joystick jacks.

## The VDG And IRQ Interrupts

The VDG, shown in Fig. 10-1, receives its operating mode from PIA 2, bits PB3 - PB7. These bits must be configured for outputting. A code written into bits 3 - 7 of the DRB will be sent to the VDG, setting its operating mode. The VDG sends out a video signal that goes to the television set to produce the picture.

The VDG also outputs two signals, HS and FS, connected to CA1 and CB1 of PIA 1. The HS signal oscillates or toggles from a high to a low and back again once every 63.5 microseconds. The FS signal oscillates or toggles from a high to a low and back again once every 16.7 milliseconds. These two signals are the sources of the IRQ interrupts. CA1 and/or CB1 can be configured via their respective control register to generate an interrupt upon an active transition of the HS and/or FS signal.

## Serial I/O

Serial I/O, or RS-232, data is transferred through PIA 2 to the serial I/O jack on the Color Computer, shown in Fig. 10-2. Data is sent out on bit PA1, where it is inverted by a device that performs the logical NOT function. The NOT component sends out a signal of a state opposite the input state. Thus, it sends out a low when a high is sent to it and it sends out a high in response to a low input. Writing a 1 bit to PA1 results in a 0 bit appearing at pin 4 of the serial I/O jack.

Input data from the serial I/O jack is read through PB0 of PIA 2. Before the data reaches PB0, it is inverted by a NOT component; whatever state is read by PB0 is actually the reverse of what is at pin 2 of the jack. Before any data is to be transmitted, PA1 must be configured for output and PB0 for input.

The READY signal is inverted and then goes to CA1 of PIA 2. CA1 should only input and may be configured via the CRA to sense a high-to-low or low-to-high transition. When a device connected to the serial I/O port becomes ready, CA1 will see a high-to-low transition. When a device becomes non-ready, CA1 will see a low-to-high transition.

The subroutine in BASIC ROM that uses the serial I/O port accommodates only printers and assumes a different signal assignment than just described. That signal assignment is: pin 1 is not used, pin 2 (RS232IN) is connected to the printer's ready signal, pin 3 is ground, and the data to be printed is sent on pin 4 (RS232OUT).

The NOT components are one-way devices. The signal can flow only into the broad side of the triangle and out the pointed end. For example, if one tries to input on PA1, one will not read the signal at pin 4 of the jack. These components also transform the electrical signal from within the computer to RS-232 standard electrical levels.

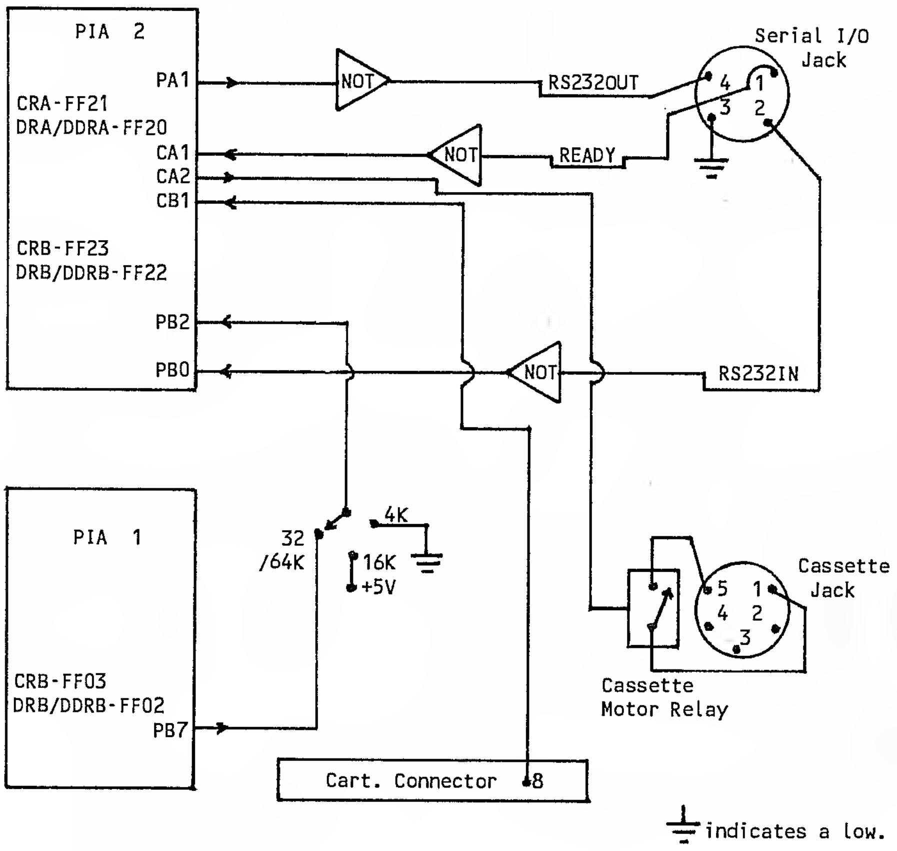


Figure 10-2
Serial I/O Port, Cassette Motor, etc.

## Cassette Motor

The cassette motor is controlled with CA2 of PIA 2. CA2 must be configured to output a high or a low to control the cassette motor relay. When CA2 sends a high, the switch in the relay closes and completes the circuit for the cassette motor. When CA2 sends a low, the switch opens and the motor stops. The switch is connected betwen pins 1 and 5 of the cassette jack, shown in Fig. 10-2.

## Memory Size Sense

The memory size is sensed with PB2 of PIA 2. PB2, configured as an input, is read after power turned on to determine the position of the memory size switch and indicate the type of memory integrated circuits within the computer. The switch is actually a connector positioned at the time of manufacture or when the memory size is changed. This switch is shown in Fig. 10-2. To read the switch position, alternating zeros and ones are sent out on PB7 of PIA 1 (PB6 in the CoCo 2). Then PB2 of PIA 1 is read several times. If PB2 is always clear, the switch is in the 4K position. If PB2 is always set, the switch is in the 16K position. If PB2 alternates between set and clear, the switch is in the 32/64K position. Based on the read switch position, the initializing program in BASIC ROM will set up the memory size control bits, M0 and M1, within the SAM.

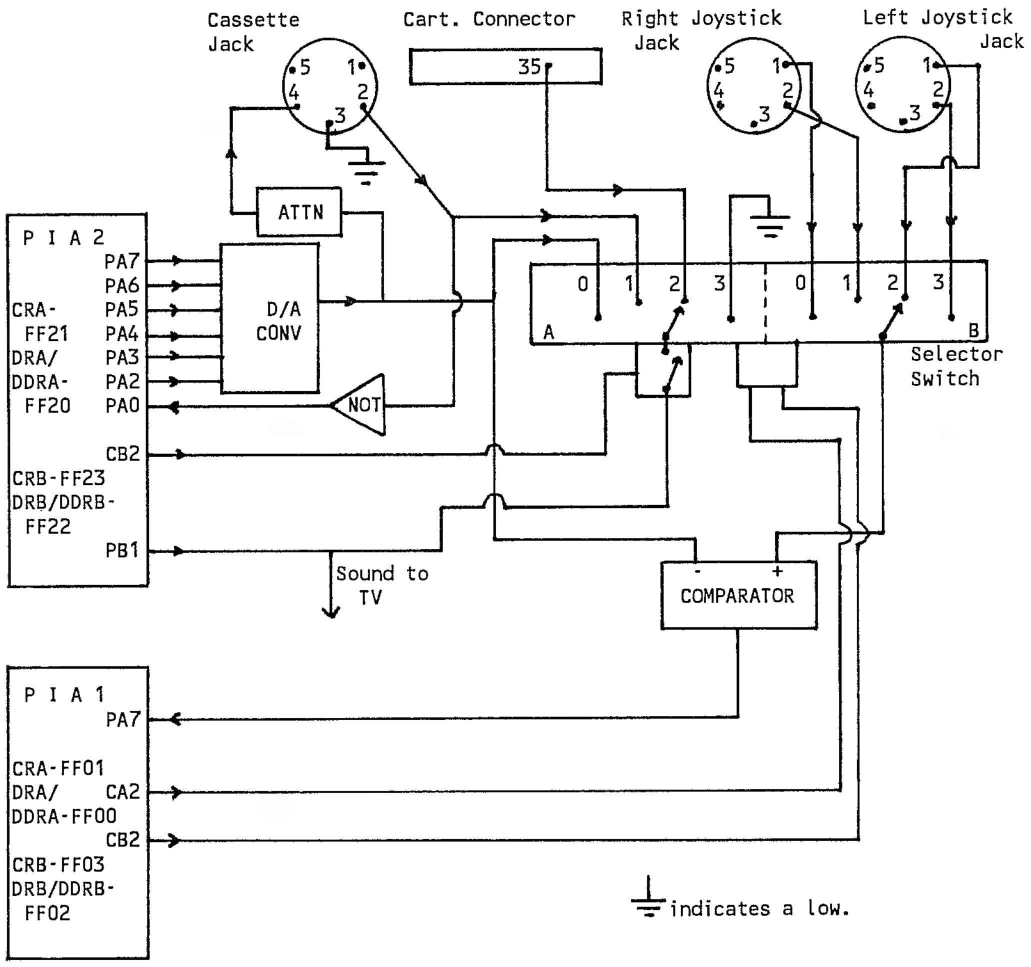


Figure 10-3
Cassette, Joysticks, and Sound Signal Flow.

**CB1 of PIA 2**

CB1 is connected to pin 8 of the cartridge connector. Since CB1 can only be an input, it can only sense a signal sent to it from the cartridge. CB1 can be configured via the CRB to sense a high-to-low or low-to-high transition and to generate an FIRQ interrupt.

**Component Operation**

Fig. 10-3 shows the components and their interconnections used to input from the cassette and the joystick positions and to output to cassette and generate sound. Many of the components may not be familiar to you, so they are described here.

■ Digital-To-Analog Converter - The digital-to-analog, or D/A, converter is connected to bits PA2 - PA7 of PIA 2. The D/A converter, upon receiving a six-bit straight binary value, generates an **analog** voltage of a magnitude proportional to the magnitude of that binary number. An analog voltage is one which can be varied by incremental amounts over a specified range. This is unlike a digital voltage, which must be at one of two levels - zero volts = a 0 bit, or +5 volts = a 1 bit.

Analog voltages are typically represented on a graph; the vertical axis represents the magnitude of the voltage and the horizontal axis represents time. Fig. 10-4 is a graph of the voltage from a nine volt battery being used to power a portable radio.

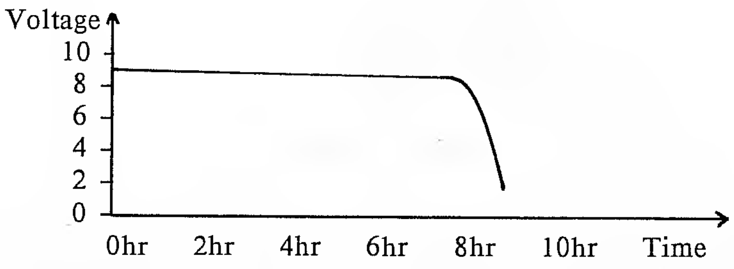


Fig. 10-4 Battery Voltage vs Time.

At time 0, the battery is fresh and generates the full nine volts. As the radio is continuously played, the battery is slowly depleted and generates a lower voltage. At about the eighth hour, the battery becomes exhausted and can only generate a diminishing voltage. Fig. 10-4 shows an analog voltage varying continuously over a period of time. The use of analog voltages becomes apparent when a waveform is used to represent something else by analogy. For example, the waveform in Fig. 10-4 could represent ones energy level thoughout the day. The vertical axis would represent the energy level and the horizontal axis would represent time. See how the energy level drops off at the end of the day?

To use the D/A converter, bits PA7 - PA2 must be configured for output. Bit PA7 is the most significant bit of the six-bit number, and PA2 is the least. The D/A converter generates a voltage, within a range of 0 to +4.5 volts, proportional to the binary value sent to it. When each bit is set, they cause the D/A converter to generate a specific voltage. The D/A converter adds together all the voltages to develop the output voltage. Table 10-1 shows the voltages that each set bit will add to the final value. If a bit is clear, it contributes nothing to the output voltage.

| Bit | Voltage |
|---|---|
| PA7 | 2.25 |
| PA6 | 1.125 |
| PA5 | 0.563 |
| PA4 | 0.281 |
| PA3 | 0.14 |
| PA2 | 0.07 |

Table 10-1 D/A Converter Bit Voltages.

To calculate the voltage generated by a six-bit value, add together the voltages each set bit generates from Table 10-1. For example, a six-bit value of 011010 would generate 1.828 volts. This is demonstrated below:

$$\begin{aligned} \text{PA6 set} &= 1.125 \\ \text{PA5 set} &= 0.563 \\ \text{PA3 set} &= \underline{0.140} \\ \text{Output voltage} &= 1.828 \end{aligned}$$

The range of output voltages is from zero, all bits clear, to +4.5, all bits set.

The D/A converter can be used to generate a waveform that varies with time by writing different values to bits 2 - 7 of the DRA of PIA 2. For example, sequentially writing the hexadecimal values 00, 20, 40, 80, and 00 generates the waveform in Fig. 10-5. The time scale of the horizontal axis depends on how often each consecutive value is written to the DRA. It could be determined by how many instructions were executed between each write to the DRA.

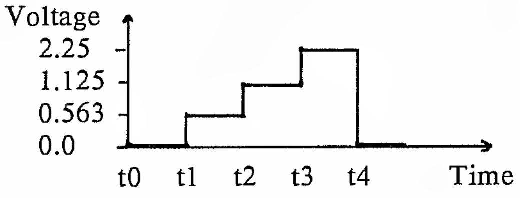


Fig. 10-5 A Waveform Generated by the D/A Converter.

■ Selector Switch - The selector switch is a dual unit composed of two four-position switches, shown in Fig. 10-3. One half is designated the A switch, and the other the B switch. The position of both switches is controlled by the CA2 and CB2 signals from PIA 1. CA2 and CB2 must be configured for output so a control code can be sent to the selector switch. The four possible combinations of the states of CA2 and CB2 put both selector switches in one of their four possible positions. The corresponding codes and switch positions are shown in Table 10-2.

| PIA 1 | | |
|---|---|---|
| CB2 | CA2 | Switch Position |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 2 |
| 1 | 1 | 3 |

2 1

Table 10-2 Selector Switch Control and Positions.

The selector switches route various analog voltages to desired destinations. Selector switch A also has a master switch controlled by CB2 of PIA 2. When the master switch is on, the analog voltage to which selector switch A is positioned will flow to its destination. When the master switch is off, no analog voltage selected by switch A is sent out. CB2 of PIA 2 must be configured to output to control the master switch. A high from CB2 turns the master switch on and a low turns it off.

■ Comparator - The comparator compares the magnitude of two analog voltages and indicates which is greater. The comparator has two inputs labelled (+) and (-). If the voltage sent to the (+) input is higher (more positive) than the voltage at the (-) input, the comparator outputs a high. If the voltage sent to the (-) input is the higher of the two, the comparator outputs a low. As presented in Fig. 10-3, the (-) input is connected to the output of the D/A converter and the (+) input is connected to the B selector switch. The B selector switch is used to select different analog voltages to be compared with the output of the D/A converter. The comparator is used in conjuction with the D/A converter to form a successive approximation analog-to-digital converter. This is described in more detail in the joystick section later in this chapter.

■ Attenuator - An attenuator is connected between the output of the D/A converter and pin 4 of the cassette jack, shown in Fig. 10-3. This is the path an analog voltage takes on its way to the cassette to be recorded. The attenuator reduces the magnitude of the range of the analog voltage going to the cassette to approximately a range of zero to plus one volt; most cassettes are designed to accept this voltage range. One could also connect pin 4 of the cassette jack to the auxillary input of a high fidelity receiver or amplifier. This way, sound generation need not be limited to the TV.

## Cassette I/O

Data is transferred to the cassette serially via pin 4 of the cassette jack, and in via pin 2. The cassette recorder is meant to record or playback an audio analog signal, an analog voltage of a frequency within the range of human hearing. The ROM subroutines, which record data, use the D/A converter to generate a voltage with a **sinusoidal waveform** to be sent to the cassette. A sinusoidal waveform is a pure or single tone. The waveform has the same shape as a graph of the sine of an angle versus the angle shown in Fig. 10-6.

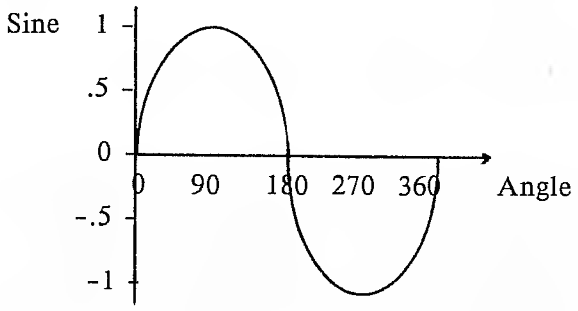
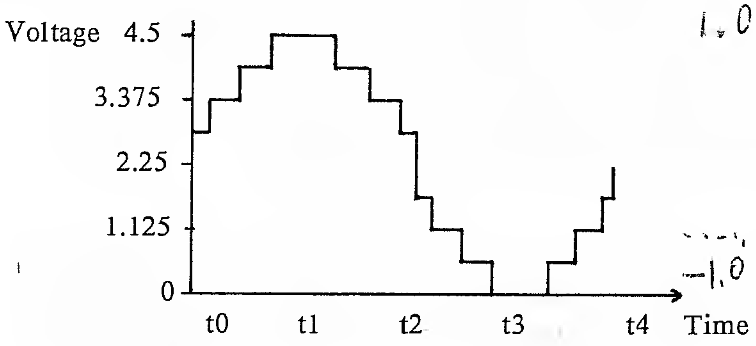


Fig. 10-6 A Plot of the Sine of an Angle vs the Angle.

The D/A converter is fed a series of values at a specific rate so it generates a waveform similar to a sine wave; the output voltage varies from 0 to +4.5 volts. Since the D/A converter can not generate a voltage at every value within its range, but only in steps of 0.07 volts, the waveform is constructed of discrete values. A typical sine wave generated by the D/A converter is shown in Fig. 10-7.



1.0

Bit zero

-1.0

Bit one

Fig. 10-7 Sine Wave Generated With the D/A Converter.

This figure shows the signal the BASIC ROM cassette write subroutine generates with the D/A converter and sends through the attenuator to the cassette. The period of time from t0 to t3 is one complete cycle. The length of the time period determines the pitch or frequency of the tone.

To record a bit, a voltage with a frequency of 1200 or 2400 hertz (cycles per second) is generated with the D/A converter and sent to the cassette recorder. If the bit is clear, the frequency is 1200 hertz; if the bit is set, it is 2400. A recording of data is made up of two tones; the pitch of the tone indicates the state of a bit. If the cassette is unplugged from the computer, a data file recording can be played back and you will hear the rapidly alternating tones.

Cassette data comes serially into PA0 of PIA 2 after passing through a NOT device. The analog voltage received from the cassette is a sinusoidal waveform similar to what was recorded. The difference is that the voltage varies symmetrically about zero: thus, the waveform appears similar to that in Fig. 10-7 - the voltage ranges between an equal positive and negative value. The amplitude of the received voltage is adjusted by the volume control of the cassette. The NOT device sends a low to PA0 if it sees an analog voltage greater than one volt. An analog voltage of less than one volt from the cassette results in a high being sent to PA0. PA0 must be configured for input and read frequently to see how often the playback voltage changes from greater than to less than one volt. This way the frequency can be found and the state of the bit that tone represents can be determined.

**Joystick Positions**

Each joystick sends two analog voltages that vary according to its position to the computer. One voltage from a joystick varies in proportion to its forward and backward position and the other to its left and right position. The forward/backward voltage comes in pin 2 and the right/left voltage comes in pin 1 of the joystick jack, shown in Fig. 10-3. The voltage ranges from zero, joystick full forward or leftward, to 4.5 volts, full backward or rightward. The position of a joystick is determined by selecting each output with the selector switch, and measuring the voltage. The voltage is measured by converting it to a digital value with the successive approximation analog-to-digital, or A/D, converter.

The successive approximation A/D converter is composed of the D/A converter, the comparator, and a program. An analog voltage to be converted to a digital number is sent to the (+) input of the comparator. The B side of the selector switch is used to select one of the four voltages to be digitized. Selector switch positions 0 and 1 select the left/right and forward/back voltages from the right joystick, and positions 2 and 3 select the corresponding voltages from the left joystick.

A voltage is digitized by outputting, through the D/A converter, a successively incremented voltage. After each higher voltage that is sent, the output of the comparator (PA7 of PIA 1) is read. PA7 clear indicates the voltage just sent by the D/A converter is marginally higher than the analog voltage at the (+) input of the comparator. At this point, the value written into the DRA of PIA 2, (sent to the D/A converter) is the digital equivalent of the analog voltage. The process requires that zero volts first be sent through the D/A converter to the (-) input of the comparator, and that voltage then be increased until the comparator indicates it is just a little higher than the analog voltage. This is the process of successive approximation.

Listing 10-1 is a program which reads the left/right position of the left joystick and displays a hexadecimal value that corresponds to its position in the lower right corner of the screen. The program determines a joystick position in a way very similar to the way the JOYIN ROM subroutine works. The displayed value is actually the digital equivalent of the analog voltage outputted by the joystick.

```
100 *PROGRAM NAME: LFJY
110 *THIS PROGRAM WILL DIGITIZE THE LEFT/RIGHT
120 *ANALOG VOLTAGE OF THE LEFT JOYSTICK USING
130 *SUCCESSIVE APPROXIMATION.
140 ******************************************
150 LFJY    LDA #$3C        GET CRB CODE
160         STA $FF03       SET CB2
170         LDA #$34        GET CRA CODE
180         STA $FF01       CLEAR CA2
190 LFA     CLRB            GET 0 VOLTS
200 LFB     STB $FF20       SEND TO D/A
210         LDA $FF00       READ COMP OUTPUT
220         BPL LFC         BRANCH IF PA7=0
230         ADDB #$04       INC VOLTAGE
240         CMPB #$FC       MAX VALUE ?
250         BEQ LFC         FORCE END IF MAX
260         BRA LFB         DO AGAIN
270 LFC     LDA #$04        GET SHIFT AMOUNT
280         MUL             MS IN A,LS IN B
290         ADDA #$70       CONVERT TO VIDEO
300         STA $5FE        PUT ON SCREEN
310         LDA #$10        SHIFT AMOUNT
320         MUL             LS IN A
330         CMPA #$09       GREATER THAN 9?
340         BHI LFD         BRANCH IF SO
350         ADDA #$70       CONV TO VIDEO
360         STA $5FF        PUT ON SCREEN
370         BRA LFA         DO AGAIN
380 LFD     ADDA #$37       CONV TO VIDEO
390         STA $5FF        PUT ON SCREEN
400         BRA LFA         DO AGAIN
410 ******************************************
420         END
```

Listing 10-1 The LFJY Program.

Lines 150 - 180 set CB2 and clear CA2, selecting position 2 of the selector switch. Lines 190 - 240 are the successive approximation loop. The

B register is first cleared and written to the D/A converter. If PA7 of PIA 1 is clear at lines 210 and 220 the conversion process is over. If not, the value in bits 7 - 2 of the B register is incremented by one by adding four to the contents of B at line 230. Lines 250 - 370 convert the six-bit value in B to two video codes for presentation on the screen. The six bits in B, bits 7 - 2, are shifted to the lower six bits so the range of the hexadecimal value is from 00 to $3F. Assemble the source code with the A/IM/WE command and verify there are no errors. Go to ZBUG and run it by entering: GLFJY. You will see two hexadecimal digits at the lower right corner of the screen vary as the left joystick is moved from side to side.

Other devices can also be connected to a joystick jack and their output voltage digitized. The successive approximation technique in Listing 10-1 resolves an analog voltage to six bits of accuracy. Thus, the digital value of a voltage is accurate to within approximately 0.07 volts.

## Sound

Sound through the TV can come from four sources: the D/A converter, the cassette, a cartridge, and PB1 of PIA 2. Selector switch A selects one of the first three sources. Position 3 of selector switch A is connected to a low and is used when no sound is desired. This can be seen in Fig. 10-3.

Sound from the D/A converter is selected by setting selector switch A to position 0 and turning on the master switch by setting CB2 of PIA 2 high. A waveform is generated by sending successive digital values to the D/A converter. The shape of the waveform determines the type of sound. One should consult an introductory physics book to learn about the characteristics of various sounds. The output, through selector switch A and the master switch, is sent to the TV and heard from its loudspeaker. One shouldn't forget to turn up the volume on the TV.

With selector switch A in position 1 and the master switch on, the playback signal from the cassette is routed to the TV loudspeaker. Thus you can hear a data file or any other recording as it is being played. A sound source in a cartridge can be selected by setting the selector switch to position 2 and turning on the master switch. A cartridge may contain a sound or voice synthesizer, greatly increasing the sound capabilities of the Color Computer. The analog signal comes from pin 35 of the cartridge connector.

A sound can also be generated by outputting successive highs and lows from PB1 of PIA 2. The sound types that can be generated this way are more limited than what the D/A converter can generate; only a high or low can be sent out of PB1. The rate at which alternating highs and lows are sent will determine the pitch of the sound. The relative length of time the output is high compared to when it is low will determine the timbre of the sound. If the master switch of selector switch A is turned on the sound will be a combination of that selected by switch A and that generated by PB1. Since PB1 is a sound source, it must be configured for outputting.

Listing 10-2 generates a tone which repeatedly slides from a low to a high pitch. The tone is generated with the PB1 sound source.

```
100 *PROGRAM NAME: SLTON
110 *THIS PROGRAM WILL REPETITIVELY GENERATE A
120 *SLIDING TONE. EACH PITCH WILL BE OUTPUTTED FOR
130 *10 CYCLES, THEN THE PITCH WILL BE INCREASED.
140 *THE SOUND IS GENERATED WITH PB1 OF PIA #2.
150 ************************************************
160 SLTON   LDA $FF23       GET CRB
170         ANDA #$FB       CLR BIT 2
180         STA $FF23       ENABLE DDRB ACCESS
190         LDA #$02        SET PB1 TO OUTPUT
200         STA $FF22       PUT IN DDRB
210         LDA $FF23       GET CRB
220         ORA #$04        SET BIT 2
230         STA $FF23       ENABLE DRB ACCESS
240 SLA     LDX #$00A0      GET MAX DELAY
250         LDB #20         GET CYCLE CNT
260         LDA $FF22       READ DRB
270 SLB     EORA #$02       TOGGLE BIT 1
280         STA $FF22       PUT IN DRB
290         DECB            COUNT CYCLES
300         BNE SLC         JMP IF NOT ZERO
310         LDB #20         GET CYCLE CNT
320         LEAX -2,X       SHORTEN DELAY
330         CMPX #$0000     HIGHEST PITCH?
340         BEQ SLA         GO START OVER
350 SLC     TFR X,Y         GET DELAY
360 SLD     LEAY -1,Y       DECR DELAY
370         CMPY #$0000     END OF DELAY?
380         BNE SLD         LOOP IF NOT
390         BRA SLB         GO TOGGLE BIT
400 ************************************************
410         END
```

Listing 10-2 The SLTON Program.

Lines 160 - 180 clear bit 2 of the CRB to allow access to the DDRB. Lines 190 - 200 configure PB1 for output. Lines 210 - 230 set bit 2 of the CRB to allow access to the DRB. Sound is generated by reversing the state of bit 1 in the DRB at a specified rate. At line 250 the B register is loaded with the number of times the bit will be reversed for each pitch. At line 240 the X register is loaded with a count used to establish a time delay between reversing PB1. Lines 270 and 280 reverse the state of PB1. Lines 350 - 390 create the time delay between reversing the state of PB1, based on the value in the X register. Lines 290 - 340 check to see if ten cycles have been produced, and if so, the value in X is decremented to reduce the time delay between reversing the state of PB1, raising the pitch.

## THE CARTRIDGE CONNECTOR

The cartridge connector is a 40 contact electrical connector on the right side of the Color Computer. Each contact, or pin, is internally connected to an electrical signal within the computer. The signals comprise the data bus, address bus, part of the control bus, and other signals. (See Chapter 3 for a refresher on the internal buses the MC6809E MPU uses.) The following descriptions are quite technical and may only be of interest to those familar with digital electronics.

The 40 contacts are arranged in two rows, each identified by its number. The numbering scheme is shown in Fig. 10-8. The connector is shown in Fig. 10-8 as viewed from outside the computer.

```
                        Top Row

39 37 35 33 31 29 27 25 23 21 19 17 15 13 11  9  7  5  3  1
 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -

 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
40 38 36 34 32 30 28 26 24 22 20 18 16 14 12 10  8  6  4  2

                       Bottom Row
```

Fig. 10-8 Cartridge Connector Contact Numbering.

Many cartridge connector signals are bidirectional; that is, a signal may be sent to or from a cartridge. Other signals are unidirectional, since they may be sent only from the computer to a cartridge or only from a cartridge to the computer. The state of a digital signal is determined by the voltage on its connector. A set state is indicated by a high, about +5 volts, and a clear state is indicated by a low, about zero volts. Thus, the signal D0, data bus bit 0, is high when bit 0 is set. Other signals may indicate other conditions, such as an interrupt signal going to the MPU. Some of these signals are high when active and others are low when active. For example, the $\overline{\text{NMI}}$ signal is active when low, meaning the MPU senses an NMI interrupt when this signal is low. When the $\overline{\text{NMI}}$ signal is high, no interrupt is sent to the MPU. Thus, the $\overline{\text{NMI}}$ signal is inactive when high. Table 10-3 lists all the signals available at the cartridge connector. Pins 1 and 2 are not connected to anything in the Color Computer 2. Those signals not bidirectional have a suffix to indicate their direction. A right arrow indicates the signal originates in the computer and is sent to the cartridge. A left arrow indicates the signal originates in the cartridge and is sent to the computer.

| Pin | Signal Name | Pin | Signal Name |
|---|---|---|---|
| 1 | -12 volts > | 2 | +12 volts > |
| 3 | $\overline{\text{HALT}}$ < | 4 | $\overline{\text{NMI}}$ < |
| 5 | $\overline{\text{RESET}}$ > | 6 | E > |
| 7 | Q > | 8 | $\overline{\text{CART}}$ < |
| 9 | +5 volts > | 10 | D0 |
| 11 | D1 | 12 | D2 |
| 13 | D3 | 14 | D4 |
| 15 | D5 | 16 | D6 |
| 17 | D7 | 18 | R/$\overline{\text{W}}$ |
| 19 | A0 | 20 | A1 |
| 21 | A2 | 22 | A3 |
| 23 | A4 | 24 | A5 |
| 25 | A6 | 26 | A7 |
| 27 | A8 | 28 | A9 |
| 29 | A10 | 30 | A11 |
| 31 | A12 | 32 | $\overline{\text{CTS}}$ > |
| 33 | GND | 34 | GND |
| 35 | SND < | 36 | $\overline{\text{SCS}}$ > |
| 37 | A13 | 38 | A14 |
| 39 | A15 | 40 | $\overline{\text{SLENB}}$ < |

Table 10-3 Cartridge Signals.

**Power**

Electrical power is sent to the cartridge through pins 1, 2, 9, 33, and 34. Pins 33 and 34 are ground connections. Ground is the reference point to which all voltages are measured. Thus, a signal of +5 volts is five volts higher than ground. Through pin 9 is sent +5 volts to provide power to components in a cartridge. Up to 300 milliamps may be drawn from the +5 volt supply. Through pin 1 is sent -12 volts, and up to 100 milliamps may be drawn. Through pin 2 is sent +12 volts, and up to 300 milliamps may be drawn. A milliamp is one thousandth of an amp or ampere. Pins 1 and 2 are not used in the Color Computer 2.

**Data Bus**

The data bus, bits D0 - D7, is available at the cartridge connector. This is a bidirectional bus; a byte can be sent to or from the cartridge. For example, a cartridge that contains ROM, such as EDTASM+, sends the contents of a memory location into the computer over the data bus. If a cartridge contains RAM or a PIA, a byte written to it would be sent to it over the data bus.

## Address Bus

The address bus, bits A0 - A15, is available at the cartridge connector. This, too, is a bidirectional bus. A memory location of ROM in a cartridge is specified by the MPU, sending that address over the address bus to the cartridge. The cartridge could also send an address to the computer to specify a RAM location to access. However, when the cartridge is using the address bus in this way, the MPU must be stopped by putting a low on the $\overline{\text{HALT}}$ signal line. The MPU will stop upon completion of the currently executing instruction and the buses will be available for use by the cartridge.

## Control Bus

The available signals of the control bus are; $\overline{\text{HALT}}$, $\overline{\text{NMI}}$, $\overline{\text{RESET}}$, E, Q, and R/$\overline{\text{W}}$. The $\overline{\text{RESET}}$ signal goes active low momentarily when the computer is turned on and when the Reset button is pushed. This signal initializes the MPU and PIAs within the computer and can be used similarly for components in a cartridge. The E and Q signals are the clock signals which originate within the computer and can be used to synchronize the operation of components in a cartridge. The R/$\overline{\text{W}}$ signal is used by whatever component is to write to or read from an address. Within the computer, only the MPU generates this signal. However, when the MPU is stopped, a component in a cartridge can use this signal to access RAM, ROM, and/or the PIAs. The $\overline{\text{NMI}}$ signal must originate in the cartridge; normally $\overline{\text{NMI}}$ is high. Setting it low momentarily causes the MPU to start the $\overline{\text{NMI}}$ interrupt sequence, if it is in a condition to do so.

The $\overline{\text{HALT}}$ signal originates in a cartridge, to be sent to the MPU. When $\overline{\text{HALT}}$ is inactive high, the MPU is continuously operating. When it is set low, the MPU stops on completion of the currently executing instruction. When the MPU is stopped it ceases to use the data and address buses and the R/$\overline{\text{W}}$ signal, allowing components in a cartridge to use these signals to access ROM, RAM, and/or PIAs in the computer. Setting $\overline{\text{HALT}}$ back high causes the MPU to resume operation at the following instruction.

## Other Signals

Other signals available at the cartridge connector are: SND, $\overline{\text{CART}}$, $\overline{\text{CTS}}$, $\overline{\text{SCS}}$, and $\overline{\text{SLENB}}$. $\overline{\text{CTS}}$ and $\overline{\text{SCS}}$ are outputs from the computer to a cartridge and the others are inputs to the computer.

$\overline{\text{CTS}}$ goes low when the MPU accesses any address from \$C000 - \$FEFF and the SAM TY control bit is clear. Under these conditions the MPU would be accessing ROM or RAM in the cartridge. Thus, $\overline{\text{CTS}}$ low enables or selects the ROM or RAM within a cartridge for transfer of a byte of data.

$\overline{\text{SCS}}$ goes low when the MPU accesses any of the dedicated addresses from \$FF40 - \$FF5F. $\overline{\text{SCS}}$ low enables or selects a PIA or similar device within a cartridge for transfer of a byte of data. $\overline{\text{SCS}}$ also goes low when the SAM TY control bit is clear and the MPU accesses any of the redundant dedicated addresses associated with addresses $FF40 - $FF5F.

SND is an analog signal that originates in a cartridge. The SND signal may be heard by setting the selector switch to position 2 and turning on the master switch. The SND signal may range from 0 to +5 volts.

The $\overline{\text{CART}}$ signal originates in a cartridge and goes to CB1 of PIA 2. If CB1 is properly configured, an FIRQ interrupt will be generated when $\overline{\text{CART}}$ goes low.

$\overline{\text{SLENB}}$ is a signal that originates in a cartridge and goes to the SAM decoding circuits in the computer. Normally $\overline{\text{SLENB}}$ is high, allowing the SAM to decode addresses, as described in Chapter Nine. When $\overline{\text{SLENB}}$ is low, all address decoding is inhibited. In this condition, no RAM, ROM or PIAs may be accessed by the MPU. The MPU would only be able to access devices in the cartridge via the data and address busses.

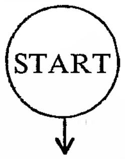
# APPENDIX A
# Flowcharts and State Diagrams

Before a program is written it should be described with a textual description, a list of specifications, and/or a mental thought. A helpful step is to draw a **flowchart** or **state diagram.** Each is a graphical representation of the sequence of operations to be performed which satisfy the program requirements. A flowchart or state diagram is very helpful when writing, testing, and/or debugging a program; it presents an easily understandable description closely related to the actual logic the program follows. Save your flowcharts or state diagrams to use as a guide if you later need to modify a program.

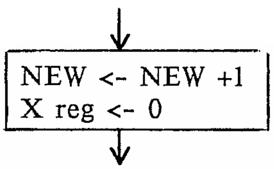
## FLOWCHARTS

A flowchart is a graphical representation of a program or section of a program. Within a flowchart major computations and the order in which they are to be performed appear. The computations are described by boxes of various shapes. The boxes are connected by arrows showing the order in which the computations will be performed.

A large circle is used to denote the beginning and the end of a program. A beginning circle is shown below; an end circle would contain the word END:

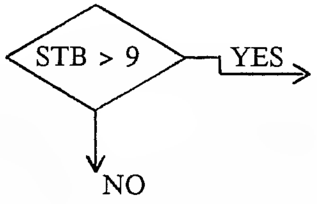


A rectangular box contains computations and shows an assignment of a value to a variable. The variable could be a memory location, specified by its symbol, or a register in the MPU. Two or more computations may be shown in a box where the top computation is to be performed first. An example is shown below.

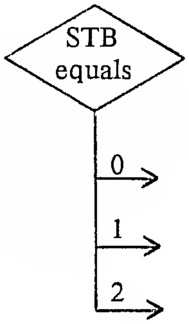


In this box above, first the value of NEW is incremented by one, then the X register is cleared. The computation to the right of the arrow within a box is performed and its result is directed to the variable to the left of the arrow.

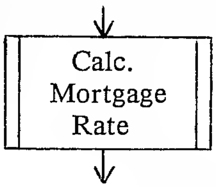
A decision is indicated by a diamond shaped box. Within the box is a condition which, if false, will cause an exit from one corner of the diamond. If the condition is true, one exits via a different corner. This is shown below:



If the value of STB is greater than nine, the YES path is taken; otherwise the NO path is taken. A single decision box can also have multiple exits:

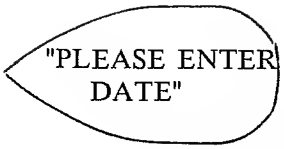


A subroutine is represented by a box with two lines on each side. The operation of the subroutine should be shown in a flowchart located on a following page. The subroutine box is shown below:

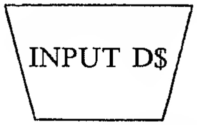


This box can also represent an interrupt handler or any other predefined process.

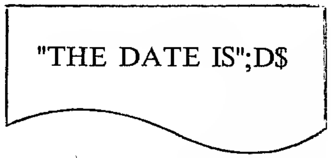
Special boxes represent input/output operations. To show outputting text to the screen, the following box is used:



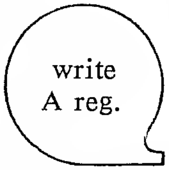
The following box represents an opereation geared to the speed of a human operator. For example, this box represents an input operation, i.e., waiting for an operator to enter the date:



Outputting to a printer is represented with a shape similar to a page of paper with the bottom torn off:



Magnetic tape or cassette I/O is represented as below:



In this example the contents of the A register are written to tape.

When a flowchart does not fit on one page, it is connected to the next page with a small circle:

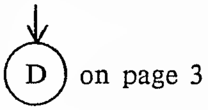
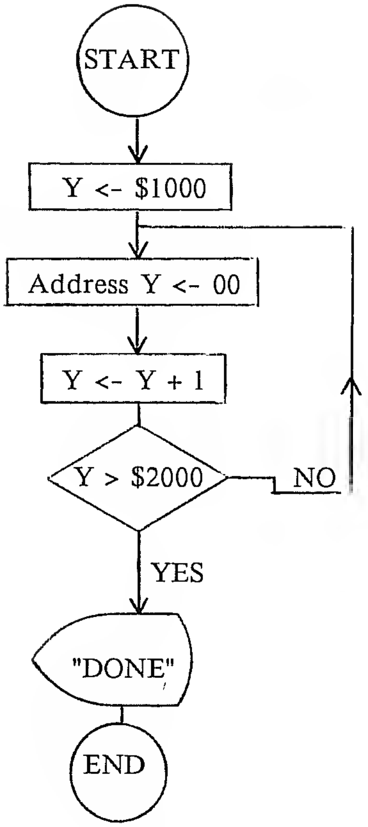


Fig. A-1 shows a flowchart of a program which clears memory locations $1000 through $2000.



Fig, A-1 Example Flowchart.

## STATE DIAGRAMS

A state diagram represents the sequence of operations a program performs as paths which go from state to state. Each state is represented by a circle in which is described the operation performed in that state. The paths (arrows) indicate the flow from one state to another under specific conditions. Thus, paths exiting a state indicate the different results of an operation performed in that state. Each path is labelled according to the result which will cause that path to be taken. Fig. A-2 shows an example of a state diagram for a program to input a decimal integer from the keyboard. The operator indicates the number is complete by hitting the ENTER key.

The keyboard is read in the first (topmost) state. The key which is depressed will determine which path to take. If the depressed key is neither a decimal digit nor the ENTER key, the read keyboard state is re-entered. If the depressed key is a decimal digit, the ASCII code of the digit is put into the buffer area. If the ENTER key is depressed, the final state is entered: the ENTER ASCII code is appended to the decimal integer in the buffer. The program section that follows this will read the digits from the buffer until the ENTER code is detected.

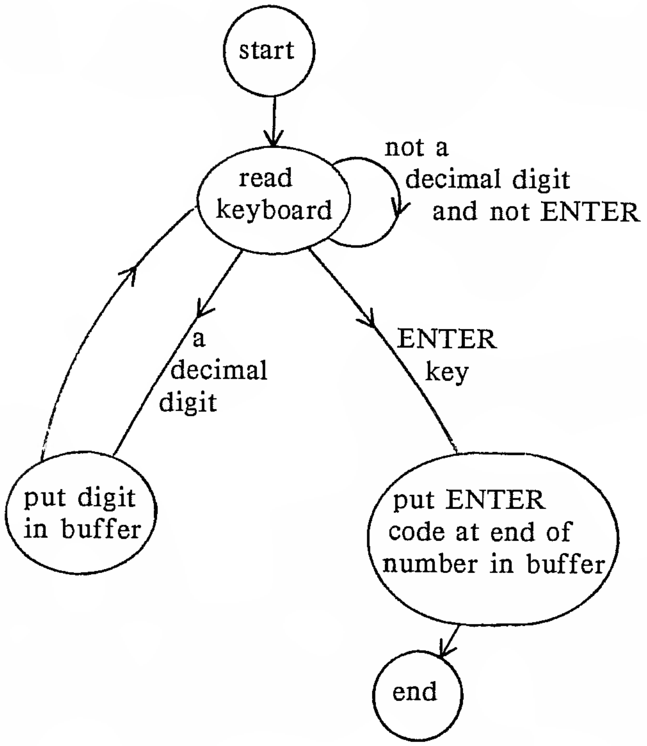
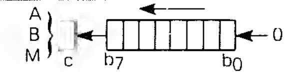
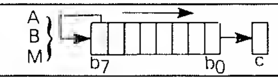

start
read keyboard
not a decimal digit and not ENTER
a decimal digit
ENTER key
put digit in buffer
put ENTER code at end of number in buffer
end

Fig A-2 Example State Diagram.

# APPENDIX B - MC6809E Instructions

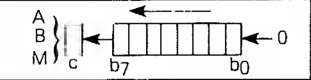
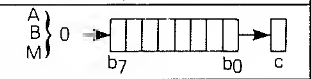
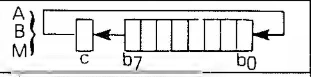
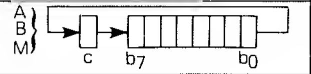
Courtesy of Motorola, Inc.

| | | Addressing Modes | | | | | | | | | | | | | | | | 5 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Immediate | | | Direct | | | Indexed | | | Extended | | | Inherent | | | | | | | | |
| Instruction | Forms | Op | ~ | # | Op | ~ | # | Op | ~ | # | Op | ~ | # | Op | ~ | # | Description | H | N | Z | V | C |
| ABX | | | | | | | | | | | | | | 3A | 3 | 1 | B + X → X (Unsigned) | • | • | • | • | • |
| ADC | ADCA | B9 | 2 | 2 | 99 | 4 | 2 | A9 | 4 + | 2 + | B9 | 5 | 3 | | | | A + M + C → A | ↕ | ↕ | ↕ | ↕ | ↕ |
| | ADCB | C9 | 2 | 2 | D9 | 4 | 2 | E9 | 4 + | 2 + | F9 | 5 | 3 | | | | B + M + C → B | ↕ | ↕ | ↕ | ↕ | ↕ |
| ADD | ADDA | 8B | 2 | 2 | 9B | 4 | 2 | AB | 4 + | 2 + | BB | 5 | 3 | | | | A + M → A | ↕ | ↕ | ↕ | ↕ | ↕ |
| | ADDB | CB | 2 | 2 | DB | 4 | 2 | EB | 4 + | 2 + | FB | 5 | 3 | | | | B + M → B | ↕ | ↕ | ↕ | ↕ | ↕ |
| | ADDD | C3 | 4 | 3 | D3 | 6 | 2 | E3 | 6 + | 2 + | F3 | 7 | 3 | | | | D + M M + 1 → D | • | ↕ | ↕ | ↕ | ↕ |
| AND | ANDA | 84 | 2 | 2 | 94 | 4 | 2 | A4 | 4 + | 2 + | B4 | 5 | 3 | | | | A ∧ M → A | • | ↕ | ↕ | 0 | • |
| | ANDB | C4 | 2 | 2 | D4 | 4 | 2 | E4 | 4 + | 2 + | F4 | 5 | 3 | | | | B ∧ M → B | • | ↕ | ↕ | 0 | • |
| | ANDCC | 1C | 3 | 2 | | | | | | | | | | | | | CC ∧ IMM → CC | | | | | 7 |
| ASL | ASLA | | | | | | | | | | | | | 4B | 2 | 1 |  | B | ↕ | ↕ | ↕ | ↕ |
| | ASLB | | | | | | | | | | | | | 58 | 2 | 1 | | B | ↕ | ↕ | ↕ | ↕ |
| | ASL | | | | 08 | 6 | 2 | 6B | 6 + | 2 + | 7B | 7 | 3 | | | | | B | ↕ | ↕ | ↕ | ↕ |
| ASR | ASRA | | | | | | | | | | | | | 47 | 2 | 1 |  | B | ↕ | ↕ | • | ↕ |
| | ASRB | | | | | | | | | | | | | 57 | 2 | 1 | | B | ↕ | ↕ | • | ↕ |
| | ASR | | | | 07 | 6 | 2 | 67 | 6 + | 2 + | 77 | 7 | 3 | | | | | B | ↕ | ↕ | • | ↕ |
| BIT | BITA | B5 | 2 | 2 | 95 | 4 | 2 | A5 | 4 + | 2 + | B5 | 5 | 3 | | | | Bit Test A (M ∧ A) | • | ↕ | ↕ | 0 | • |
| | BITB | C5 | 2 | 2 | D5 | 4 | 2 | E5 | 4 + | 2 + | F5 | 5 | 3 | | | | Bit Test B (M ∧ B) | • | ↕ | ↕ | 0 | • |
| CLR | CLRA | | | | | | | | | | | | | 4F | 2 | 1 | 0 → A | • | 0 | 1 | 0 | 0 |
| | CLRB | | | | | | | | | | | | | 5F | 2 | 1 | 0 → B | • | 0 | 1 | 0 | 0 |
| | CLR | | | | 0F | 6 | 2 | 6F | 6 + | 2 + | 7F | 7 | 3 | | | | 0 → M | • | 0 | 1 | 0 | 0 |
| CMP | CMPA | B1 | 2 | 2 | 91 | 4 | 2 | A1 | 4 + | 2 + | B1 | 5 | 3 | | | | Compare M from A | 8 | ↕ | ↕ | ↕ | ↕ |
| | CMPB | C1 | 2 | 2 | D1 | 4 | 2 | E1 | 4 + | 2 + | F1 | 5 | 3 | | | | Compare M from B | 8 | ↕ | ↕ | ↕ | ↕ |
| | CMPD | 10<br>B3 | 5 | 4 | 10<br>93 | 7 | 3 | 10<br>A3 | 7 + | 3 + | 10<br>B3 | 8 | 4 | | | | Compare M M + 1 from D | • | ↕ | ↕ | ↕ | ↕ |
| | CMPS | 11<br>8C | 5 | 4 | 11<br>9C | 7 | 3 | 11<br>AC | 7 + | 3 + | 11<br>BC | 8 | 4 | | | | Compare M M + 1 from S | • | ↕ | ↕ | ↕ | ↕ |
| | CMPU | 11<br>B3 | 5 | 4 | 11<br>93 | 7 | 3 | 11<br>A3 | 7 + | 3 + | 11<br>B3 | 8 | 4 | | | | Compare M M + 1 from U | • | ↕ | ↕ | ↕ | ↕ |
| | CMPX | BC | 4 | 3 | 9C | 6 | 2 | AC | 6 + | 2 + | BC | 7 | 3 | | | | Compare M M + 1 from X | • | ↕ | ↕ | ↕ | ↕ |
| | CMPY | 10<br>BC | 5 | 4 | 10<br>9C | 7 | 3 | 10<br>AC | 7 + | 3 + | 10<br>BC | 8 | 4 | | | | Compare M M + 1 from Y | • | ↕ | ↕ | ↕ | ↕ |
| COM | COMA | | | | | | | | | | | | | 43 | 2 | 1 | $\overline{A}$ → A | • | ↕ | ↕ | 0 | 1 |
| | COMB | | | | | | | | | | | | | 53 | 2 | 1 | $\overline{B}$ → B | • | ↕ | ↕ | 0 | 1 |
| | COM | | | | 03 | 6 | 2 | 63 | 6 + | 2 + | 73 | 7 | 3 | | | | $\overline{M}$ → M | • | ↕ | ↕ | 0 | 1 |

| | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CWAI | | 3C | ≥20 | 2 | | | | | | | | | | | | | CC Λ IMM→CC Wait for Interrupt | | | | | 7 |
| DAA | | | | | | | | | | | | | | 19 | 2 | 1 | Decimal Adjust A | • | ↕ | ↕ | 0 | ↕ |
| DEC | DECA | | | | | | | | | | | | | 4A | 2 | 1 | A - 1→A | • | ↕ | ↕ | ↕ | • |
| | DEC8 | | | | | | | | | | | | | 5A | 2 | 1 | B - 1→8 | • | ↕ | ↕ | ↕ | • |
| | DEC | | | | 0A | 6 | 2 | 6A | 6+ | 2+ | 7A | 7 | 3 | | | | M - 1→M | • | ↕ | ↕ | ↕ | • |
| EOR | EORA | 88 | 2 | 2 | 98 | 4 | 2 | A8 | 4+ | 2+ | 88 | 5 | 3 | | | | A ⊻ M→A | • | ↕ | ↕ | 0 | • |
| | EOR8 | C8 | 2 | 2 | D8 | 4 | 2 | E8 | 4+ | 2+ | F8 | 5 | 3 | | | | B ⊻ M→B | • | ↕ | ↕ | 0 | • |
| EXG | R1, R2 | 1E | 8 | 2 | | | | | | | | | | | | | R1→R2$^2$ | • | • | • | • | • |
| INC | INCA | | | | | | | | | | | | | 4C | 2 | 1 | A + 1→A | • | ↕ | ↕ | ↕ | • |
| | INC8 | | | | | | | | | | | | | 5C | 2 | 1 | 8 + 1→8 | • | ↕ | ↕ | ↕ | • |
| | INC | | | | 0C | 6 | 2 | 6C | 6+ | 2+ | 7C | 7 | 3 | | | | M + 1→M | • | ↕ | ↕ | ↕ | • |
| JMP | | | | | 0E | 3 | 2 | 6E | 3+ | 2+ | 7E | 4 | 3 | | | | EA$^3$→PC | • | • | • | • | • |
| JSR | | | | | 9D | 7 | 2 | AD | 7+ | 2+ | 8D | 8 | 3 | | | | Jump to Subroutine | • | • | • | • | • |
| LD | LDA | 86 | 2 | 2 | 96 | 4 | 2 | A6 | 4+ | 2+ | 86 | 5 | 3 | | | | M→A | • | ↕ | ↕ | 0 | • |
| | LDB | C6 | 2 | 2 | D6 | 4 | 2 | E6 | 4+ | 2+ | F6 | 5 | 3 | | | | M→8 | • | ↕ | ↕ | 0 | • |
| | LDD | CC | 3 | 3 | DC | 5 | 2 | EC | 5+ | 2+ | FC | 6 | 3 | | | | M:M + 1→D | • | ↕ | ↕ | 0 | • |
| | LDS | 10 CE | 4 | 4 | 10 DE | 6 | 3 | 10 EE | 6+ | 3+ | 10 FE | 7 | 4 | | | | M:M + 1→S | • | ↕ | ↕ | 0 | • |
| | LDU | CE | 3 | 3 | DE | 5 | 2 | EE | 5+ | 2+ | FE | 6 | 3 | | | | M:M + 1→U | • | ↕ | ↕ | 0 | • |
| | LDX | 8E | 3 | 3 | 9E | 5 | 2 | AE | 5+ | 2+ | 8E | 6 | 3 | | | | M:M + 1→X | • | ↕ | ↕ | 0 | • |
| | LDY | 10 8E | 4 | 4 | 10 9E | 6 | 3 | 10 AE | 6+ | 3+ | 10 8E | 7 | 4 | | | | M:M + 1→Y | • | ↕ | ↕ | 0 | • |
| LEA | LEAS | | | | | | | 32 | 4+ | 2+ | | | | | | | EA$^3$→S | • | • | • | • | • |
| | LEAU | | | | | | | 33 | 4+ | 2+ | | | | | | | EA$^3$→U | • | • | • | • | • |
| | LEAX | | | | | | | 30 | 4+ | 2+ | | | | | | | EA$^3$→X | • | • | ↕ | • | • |
| | LEAY | | | | | | | 31 | 4+ | 2+ | | | | | | | EA$^3$→Y | • | • | ↕ | • | • |

LEGEND:

| | | | | | |
|---|---|---|---|---|---|
| OP | Operation Code (Hexadecimal) | $\overline{M}$ | Complement of M | ↕ | Test and set if true, cleared otherwise |
| ~ | Number of MPU Cycles | → | Transfer Into | • | Not Affected |
| # | Number of Program Bytes | H | Half-carry (from bit 3) | CC | Condition Code Register |
| + | Arithmetic Plus | N | Negative (sign bit) | : | Concatenation |
| − | Arithmetic Minus | Z | Zero result | V | Logical or |
| • | Multiply | V | Overflow, 2's complement | Λ | Logical and |
| | | C | Carry from ALU | ⊻ | Logical Exclusive or |

| Instruction | Forms | Addressing Modes | | | | | | | | | | | | | | | Description | 5 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Immediate | | | Direct | | | Indexed$^1$ | | | Extended | | | Inherent | | | | | | | | |
| | | Op | ~ | # | Op | ~ | # | Op | ~ | # | Op | ~ | # | Op | ~ | # | | H | N | Z | V | C |
| LSL | LSLA | | | | | | | | | | | | | 48 | 2 | 1 |  | • | ↕ | ↕ | ↕ | ↕ |
| | LSLB | | | | | | | | | | | | | 58 | 2 | 1 | | • | ↕ | ↕ | ↕ | ↕ |
| | LSL | | | | 0B | 6 | 2 | 6B | 6+ | 2+ | 78 | 7 | 3 | | | | | • | ↕ | ↕ | ↕ | ↕ |
| LSR | LSRA | | | | | | | | | | | | | 44 | 2 | 1 |  | • | 0 | ↕ | • | ↕ |
| | LSRB | | | | | | | | | | | | | 54 | 2 | 1 | | • | 0 | ↕ | • | ↕ |
| | LSR | | | | 04 | 6 | 2 | 64 | 6+ | 2+ | 74 | 7 | 3 | | | | | • | 0 | ↕ | • | ↕ |
| MUL | | | | | | | | | | | | | | 3D | 11 | 1 | A × B → D (Unsigned) | • | • | ↕ | • | 9 |
| NEG | NEGA | | | | | | | | | | | | | 40 | 2 | 1 | $\overline{A}$ + 1 → A | B | ↕ | ↕ | ↕ | ↕ |
| | NEGB | | | | | | | | | | | | | 50 | 2 | 1 | $\overline{B}$ + 1 → B | B | ↕ | ↕ | ↕ | ↕ |
| | NEG | | | | 00 | 6 | 2 | 60 | 6+ | 2+ | 70 | 7 | 3 | | | | $\overline{M}$ + 1 → M | B | ↕ | ↕ | ↕ | ↕ |
| NOP | | | | | | | | | | | | | | 12 | 2 | 1 | No Operation | • | • | • | • | • |
| OR | ORA | BA | 2 | 2 | 9A | 4 | 2 | AA | 4+ | 2+ | BA | 5 | 3 | | | | A V M → A | • | ↕ | ↕ | 0 | • |
| | ORB | CA | 2 | 2 | DA | 4 | 2 | EA | 4+ | 2+ | FA | 5 | 3 | | | | B V M → B | • | ↕ | ↕ | 0 | • |
| | ORCC | 1A | 3 | 2 | | | | | | | | | | | | | CC V IMM → CC | | | | 7 | |
| PSH | PSHS | 34 | 5+$^4$ | 2 | | | | | | | | | | | | | Push Registers on S Stack | • | • | • | • | • |
| | PSHU | 36 | 5+$^4$ | 2 | | | | | | | | | | | | | Push Registers on U Stack | • | • | • | • | • |
| PUL | PULS | 35 | 5+$^4$ | 2 | | | | | | | | | | | | | Pull Registers from S Stack | • | • | • | • | • |
| | PULU | 37 | 5+$^4$ | 2 | | | | | | | | | | | | | Pull Registers from U Stack | • | • | • | • | • |
| ROL | ROLA | | | | | | | | | | | | | 49 | 2 | 1 |  | • | ↕ | ↕ | ↕ | ↕ |
| | ROLB | | | | | | | | | | | | | 59 | 2 | 1 | | • | ↕ | ↕ | ↕ | ↕ |
| | ROL | | | | 09 | 6 | 2 | 69 | 6+ | 2+ | 79 | 7 | 3 | | | | | • | ↕ | ↕ | ↕ | ↕ |
| ROR | RORA | | | | | | | | | | | | | 46 | 2 | 1 |  | • | ↕ | ↕ | • | ↕ |
| | RORB | | | | | | | | | | | | | 56 | 2 | 1 | | • | ↕ | ↕ | • | ↕ |
| | ROR | | | | 06 | 6 | 2 | 66 | 6+ | 2+ | 76 | 7 | 3 | | | | | • | ↕ | ↕ | • | ↕ |
| RTI | | | | | | | | | | | | | | 3B | 6/15 | 1 | Return From Interrupt | | | | | 7 |
| RTS | | | | | | | | | | | | | | 39 | 5 | 1 | Return from Subroutine | • | • | • | • | • |
| SBC | SBCA | 82 | 2 | 2 | 92 | 4 | 2 | A2 | 4+ | 2+ | B2 | 5 | 3 | | | | A − M − C → A | 8 | ↕ | ↕ | ↕ | ↕ |
| | SBCB | C2 | 2 | 2 | D2 | 4 | 2 | E2 | 4+ | 2+ | F2 | 5 | 3 | | | | B − M − C → B | 8 | ↕ | ↕ | ↕ | ↕ |
| SEX | | | | | | | | | | | | | | 1D | 2 | 1 | Sign Extend B into A | • | ↕ | ↕ | 0 | • |
| ST | STA | | | | 97 | 4 | 2 | A7 | 4+ | 2+ | B7 | 5 | 3 | | | | A → M | • | ↕ | ↕ | 0 | • |
| | STB | | | | D7 | 4 | 2 | E7 | 4+ | 2+ | F7 | 5 | 3 | | | | B → M | • | ↕ | ↕ | 0 | • |
| | STD | | | | DD | 5 | 2 | ED | 5+ | 2+ | FD | 6 | 3 | | | | D → M:M + 1 | • | ↕ | ↕ | 0 | • |
| | STS | | | | 10 | 6 | 3 | 10 | 6+ | 3+ | 10 | 7 | 4 | | | | S → M:M + 1 | • | ↕ | ↕ | 0 | • |
| | | | | | DF | | | EF | | | FF | | | | | | | | | | | |
| | STU | | | | DF | 5 | 2 | EF | 5+ | 2+ | FF | 6 | 3 | | | | U → M:M + 1 | • | ↕ | ↕ | 0 | • |
| | STX | | | | 9F | 5 | 2 | AF | 5+ | 2+ | BF | 6 | 3 | | | | X → M:M + 1 | • | ↕ | ↕ | 0 | • |
| | STY | | | | 10 | 6 | 3 | 10 | 6+ | 3+ | 10 | 7 | 4 | | | | Y → M:M + 1 | • | ↕ | ↕ | 0 | • |
| | | | | | 9F | | | AF | | | BF | | | | | | | | | | | |

| | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SUB | SUBA | 80 | 2 | 2 | 90 | 4 | 2 | A0 | 4+ | 2+ | B0 | 5 | 3 | | | | A − M → A | 8 | ↕ | ↕ | ↕ | ↕ |
| | SUBB | C0 | 2 | 2 | D0 | 4 | 2 | E0 | 4+ | 2+ | F0 | 5 | 3 | | | | B − M → B | 8 | ↕ | ↕ | ↕ | ↕ |
| | SUBD | 83 | 4 | 3 | 93 | 6 | 2 | A3 | 6+ | 2+ | B3 | 7 | 3 | | | | D − M:M + 1 → D | • | ↕ | ↕ | ↕ | ↕ |
| SWI | SWI$^6$ | | | | | | | | | | | | | 3F | 19 | 1 | Software Interrupt 1 | • | • | • | • | • |
| | SWI2$^6$ | | | | | | | | | | | | | 10<br>3F | 20 | 2 | Software Interrupt 2 | • | • | • | • | • |
| | SWI3$^6$ | | | | | | | | | | | | | 11<br>3F | 20 | 1 | Software Interrupt 3 | • | • | • | • | • |
| SYNC | | | | | | | | | | | | | | 13 | ≥4 | 1 | Synchronize to Interrupt | • | • | • | • | • |
| TFR | R1, R2 | 1F | 6 | 2 | | | | | | | | | | | | | R1 → R2$^2$ | • | • | • | • | • |
| TST | TSTA | | | | | | | | | | | | | 4D | 2 | 1 | Test A | • | ↕ | ↕ | 0 | • |
| | TSTB | | | | | | | | | | | | | 5D | 2 | 1 | Test B | • | ↕ | ↕ | 0 | • |
| | TST | | | | 0D | 6 | 2 | 6D | 6+ | 2+ | 7D | 7 | 3 | | | | Test M | • | ↕ | ↕ | 0 | • |

NOTES:

1. This column gives a base cycle and byte count. To obtain total count, add the values obtained from the INDEXED ADDRESSING MODE table, Table 2.
2. R1 and R2 may be any pair of 8 bit or any pair of 16 bit registers.
   The 8 bit registers are: A, B, CC, DP
   The 16 bit registers are: X, Y, U, S, D, PC
3. EA is the effective address.
4. The PSH and PUL instructions require 5 cycles plus 1 cycle for each **byte** pushed or pulled.
5. 5(6) means: 5 cycles if branch not taken, 6 cycles if taken (Branch instructions).
6. SWI sets I and F bits. SWI2 and SWI3 do not affect I and F.
7. Conditions Codes set as a direct result of the instruction.
8. Vaue of half-carry flag is undefined.
9. Special Case — Carry set if b7 is SET.

## Branch Instructions

| | | Addressing Mode | | | | 5 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Reletive | | | | | | | | |
| Instruction | Forms | OP | ~ 5 | # | Description | H | N | Z | V | C |
| BCC | BCC | 24 | 3 | 2 | Branch C = 0 | • | • | • | • | • |
| | LBCC | 10 | 5(6) | 4 | Long Branch | • | • | • | • | • |
| | | 24 | | | C = 0 | | | | | |
| BCS | BCS | 25 | 3 | 2 | Branch C = 1 | • | • | • | • | • |
| | LBCS | 10 | 5(6) | 4 | Long Branch | • | • | • | • | • |
| | | 25 | | | C = 1 | | | | | |
| BEO | BEO | 27 | 3 | 2 | Branch Z = 1 | • | • | • | • | • |
| | LBEO | 10 | 5(6) | 4 | Long Branch | • | • | • | • | • |
| | | 27 | | | Z = 0 | | | | | |
| BGE | BGE | 2C | 3 | 2 | Branch ≥ Zero | • | • | • | • | • |
| | LBGE | 10 | 5(6) | 4 | Long Branch ≥ Zero | • | • | • | • | • |
| | | 2C | | | | | | | | |
| BGT | BGT | 2E | 3 | 2 | Branch > Zero | • | • | • | • | • |
| | LBGT | 10 | 5(6) | 4 | Long Branch > Zero | • | • | • | • | • |
| | | 2E | | | | | | | | |
| BHI | BHI | 22 | 3 | 2 | Branch Higher | • | • | • | • | • |
| | LBHI | 10 | 5(6) | 4 | Long Branch Higher | • | • | • | • | • |
| | | 22 | | | | | | | | |
| BHS | BHS | 24 | 3 | 2 | Branch Higher | • | • | • | • | • |
| | | | | | or Same | | | | | |
| | LBHS | 10 | 5(6) | 4 | Long Branch Higher | • | • | • | • | • |
| | | 24 | | | or Same | | | | | |
| BLE | BLE | 2F | 3 | 2 | Branch ≤ Zero | • | • | • | • | • |
| | LBLE | 10 | 5(6) | 4 | Long Branch ≤ Zero | • | • | • | • | • |
| | | 2F | | | | | | | | |
| BLO | BLO | 25 | 3 | 2 | Branch lower | • | • | • | • | • |
| | LBLO | 10 | 5(6) | 4 | Long Branch Lower | • | • | • | • | • |
| | | 25 | | | | | | | | |

| | | Addressing Mode | | | | 5 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Relative | | | | | | | | |
| Instruction | Forms | OP | ~ 5 | # | Description | H | N | Z | V | C |
| BLS | BLS | 23 | 3 | 2 | Branch Lower | • | • | • | • | • |
| | | | | | or Same | | | | | |
| | LBLS | 10 | 5(6) | 4 | Long Branch Lower | • | • | • | • | • |
| | | 23 | | | or Same | | | | | |
| BLT | BLT | 2D | 3 | 2 | Branch < Zero | • | • | • | • | • |
| | LBLT | 10 | 5(6) | 4 | Long Branch < Zero | • | • | • | • | • |
| | | 2D | | | | | | | | |
| BMI | BMI | 2B | 3 | 2 | Branch Minus | • | • | • | • | • |
| | LBMI | 10 | 5(6) | 4 | Long Branch Minus | • | • | • | • | • |
| | | 2B | | | | | | | | |
| BNE | BNE | 26 | 3 | 2 | Branch Z = 0 | • | • | • | • | • |
| | LBNE | 10 | 5(6) | 4 | Long Branch | • | • | • | • | • |
| | | 26 | | | Z ≠ 0 | | | | | |
| BPL | BPL | 2A | 3 | 2 | Branch Plus | • | • | • | • | • |
| | LBPL | 10 | 5(6) | 4 | Long Branch Plus | • | • | • | • | • |
| | | 2A | | | | | | | | |
| BRA | BRA | 20 | 3 | 2 | Branch Always | • | • | • | • | • |
| | LBRA | 16 | 5 | 3 | Long Branch Always | • | • | • | • | • |
| BRN | BRN | 21 | 3 | 2 | Branch Never | • | • | • | • | • |
| | LBRN | 10 | 5 | 4 | Long Branch Never | • | • | • | • | • |
| | | 21 | | | | | | | | |
| BSR | BSR | BD | 7 | 2 | Branch to Subroutine | • | • | • | • | • |
| | LBSR | 17 | 9 | 3 | Long Branch to | • | • | • | • | • |
| | | | | | Subroutine | | | | | |
| BVC | ˀVC | 2B | 3 | 2 | Branch V = 0 | • | • | • | • | • |
| | ι BVC | 10 | 5(6) | 4 | Long Branch | • | • | • | • | • |
| | | 2B | | | V = 0 | | | | | |
| BVS | BVS | 29 | 3 | 2 | Branch V = 1 | • | • | • | • | • |
| | LBVS | 10 | 5(6) | 4 | Long Branch | • | • | • | • | • |
| | | 29 | | | V = 1 | | | | | |

SIMPLE BRANCHES

| | OP | ~ | # |
|---|---|---|---|
| BRA | 20 | 3 | 2 |
| LBRA | 16 | 5 | 3 |
| BRN | 21 | 3 | 2 |
| LBRN | 1021 | 5 | 4 |
| BSR | BD | 7 | 2 |
| LBSR | 17 | 9 | 3 |

SIMPLE CONDITIONAL BRANCHES (Notes 1-4)

| Test | True | OP | False | OP |
|---|---|---|---|---|
| N = 1 | BMI | 2B | BPL | 2A |
| Z = 1 | BEQ | 27 | BNE | 26 |
| V = 1 | BVS | 29 | BVC | 2B |
| C = 1 | BCS | 25 | BCC | 24 |

SIGNED CONDITIONAL BRANCHES (Notes 1-4)

| Test | True | OP | False | OP |
|---|---|---|---|---|
| r>m | BGT | 2E | BLE | 2F |
| r≥m | BGE | 2C | BLT | 2D |
| r=m | BEQ | 27 | BNE | 26 |
| r≤m | BLE | 2F | BGT | 2E |
| r<m | BLT | 2D | BGE | 2C |

UNSIGNED CONDITIONAL BRANCHES (Notes 1-4)

| Test | True | OP | False | OP |
|---|---|---|---|---|
| r>m | BHI | 22 | BLS | 23 |
| r≥m | BHS | 24 | BLO | 25 |
| r=m | BEQ | 27 | BNE | 26 |
| r≤m | BLS | 23 | BHI | 22 |
| r<m | BLO | 25 | BHS | 24 |

NOTES:

1. All conditional branches have both short and long variations.
2. All short branches are 2 bytes and require 3 cycles.
3. All conditional long branches are formed by prefixing the short branch opcode with $10 and using a 16-bit destination offset.
4. All conditional long branches require 4 bytes and 6 cycles if the branch is taken or 5 cycles if the branch is not taken.
5. 5(6) means: 5 cycles if branch not taken, 6 cycles if taken.

# APPENDIX C
# Color Computer Character Codes

## CODES GENERATED BY THE KEYBOARD (POLCAT)

| Key | Not Shifted | | Shifted | | Key | Not Shifted | | Shifted | |
|---|---|---|---|---|---|---|---|---|---|
| | Dec | Hex | Dec | Hex | | Dec | Hex | Dec | Hex |
| Break | 3 | 03 | 3 | 03 | B | 98 | 62 | 66 | 42 |
| <- | 8 | 08 | 21 | 15 | C | 99 | 63 | 67 | 43 |
| ->(]) | 9 | 09 | 93 | 5D | D | 100 | 64 | 68 | 44 |
| ↓([) | 10 | 0A | 91 | 5B | E | 101 | 65 | 69 | 45 |
| ↑(<-) | 94 | 5E | 95 | 5F | F | 102 | 66 | 70 | 46 |
| Clr (\) | 12 | 0C | 92 | 5C | G | 103 | 67 | 71 | 47 |
| Enter | 13 | 0D | | | H | 104 | 68 | 72 | 48 |
| Space | 32 | 20 | | | I | 105 | 69 | 73 | 49 |
| , (<) | 44 | 2C | 60 | 3C | J | 106 | 6A | 74 | 4A |
| - (=) | 45 | 2D | 61 | 3D | K | 107 | 6B | 75 | 4B |
| . (>) | 46 | 2E | 62 | 3E | L | 108 | 6C | 76 | 4C |
| / (?) | 47 | 2F | 63 | 3F | M | 109 | 6D | 77 | 4D |
| 0 | 48 | 30 | | | N | 110 | 6E | 78 | 4E |
| 1 (!) | 49 | 31 | 33 | 21 | O | 111 | 6F | 79 | 4F |
| 2 (") | 50 | 32 | 34 | 22 | P | 112 | 70 | 80 | 50 |
| 3 (#) | 51 | 33 | 35 | 23 | Q | 113 | 71 | 81 | 51 |
| 4 ($) | 52 | 34 | 36 | 24 | R | 114 | 72 | 82 | 52 |
| 5 (%) | 53 | 35 | 37 | 25 | S | 115 | 73 | 83 | 53 |
| 6 (&) | 54 | 36 | 38 | 26 | T | 116 | 74 | 84 | 54 |
| 7 (') | 55 | 37 | 39 | 27 | U | 117 | 75 | 85 | 55 |
| 8 (() | 56 | 38 | 40 | 28 | V | 118 | 76 | 86 | 56 |
| 9 ()) | 57 | 39 | 41 | 29 | W | 119 | 77 | 87 | 57 |
| : (*) | 58 | 3A | 42 | 2A | X | 120 | 78 | 88 | 58 |
| ; (+) | 59 | 3B | 43 | 2B | Y | 121 | 79 | 89 | 59 |
| @ | 64 | 40 | 19 | 13 | Z | 122 | 7A | 90 | 5A |
| A | 97 | 61 | 65 | 41 | | | | | |

Above items in parentheses are the characters generated when the SHIFT key is held down. Note that the keyboard may be in the all upper case mode when generating the alphabetic characters. Press SHIFT and 0 to change mode so that lower and upper case alphabetic characters can be generated. Repeat this procedure to change back to the all upper case mode.

## VIDEO DISPLAY CODES

| Character | Normal Video | | Reverse Video | | Character | Normal Video | | Reverse Video | |
|---|---|---|---|---|---|---|---|---|---|
| | Dec | Hex | Dec | Hex | | Dec | Hex | Dec | Hex |
| @ | 64 | 40 | 0 | 00 | space | 96 | 60 | 32 | 20 |
| A | 65 | 41 | 1 | 01 | ! | 97 | 61 | 33 | 21 |
| B | 66 | 42 | 2 | 02 | " | 98 | 62 | 34 | 22 |
| C | 67 | 43 | 3 | 03 | # | 99 | 63 | 35 | 23 |
| D | 68 | 44 | 4 | 04 | $ | 100 | 64 | 36 | 24 |
| E | 69 | 45 | 5 | 05 | % | 101 | 65 | 37 | 25 |
| F | 70 | 46 | 6 | 06 | & | 102 | 66 | 38 | 26 |
| G | 71 | 47 | 7 | 07 | ' | 103 | 67 | 39 | 27 |
| H | 72 | 48 | 8 | 08 | ( | 104 | 68 | 40 | 28 |
| I | 73 | 49 | 9 | 09 | ) | 105 | 69 | 41 | 29 |
| J | 74 | 4A | 10 | 0A | * | 106 | 6A | 42 | 2A |
| K | 75 | 4B | 11 | 0B | + | 107 | 6B | 43 | 2B |
| L | 76 | 4C | 12 | 0C | , | 108 | 6C | 44 | 2C |
| M | 77 | 4D | 13 | 0D | - | 109 | 6D | 45 | 2D |
| N | 78 | 4E | 14 | 0E | . | 110 | 6E | 46 | 2E |
| O | 79 | 4F | 15 | 0F | / | 111 | 6F | 47 | 2F |
| P | 80 | 50 | 16 | 10 | 0 | 112 | 70 | 48 | 30 |
| Q | 81 | 51 | 17 | 11 | 1 | 113 | 71 | 49 | 31 |
| R | 82 | 52 | 18 | 12 | 2 | 114 | 72 | 50 | 32 |
| S | 83 | 53 | 19 | 13 | 3 | 115 | 73 | 51 | 33 |
| T | 84 | 54 | 20 | 14 | 4 | 116 | 74 | 52 | 34 |
| U | 85 | 55 | 21 | 15 | 5 | 117 | 75 | 53 | 35 |
| V | 86 | 56 | 22 | 16 | 6 | 118 | 76 | 54 | 36 |
| W | 87 | 57 | 23 | 17 | 7 | 119 | 77 | 55 | 37 |
| X | 88 | 58 | 24 | 18 | 8 | 120 | 78 | 56 | 38 |
| Y | 89 | 59 | 25 | 19 | 9 | 121 | 79 | 57 | 39 |
| Z | 90 | 5A | 26 | 1A | : | 122 | 7A | 58 | 3A |
| [ | 91 | 5B | 27 | 1B | ; | 123 | 7B | 59 | 3B |
| \ | 92 | 5C | 28 | 1C | < | 124 | 7C | 60 | 3C |
| ] | 93 | 5D | 29 | 1D | = | 125 | 7D | 61 | 3D |
| ↑ | 94 | 5E | 30 | 1E | > | 126 | 7E | 62 | 3E |
| <- | 95 | 5F | 31 | 1F | ? | 127 | 7F | 63 | 3F |

Normal video is a black character on a light background. Reverse video is a light character on a black background.

# APPENDIX D - ASCII CODES

Most Significant Digit (Hex)

| Least Significant Digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | NUL | DLE | SP | 0 | @ | P | ` | p |
| 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 2 | STX | DC2 | " | 2 | B | R | b | r |
| 3 | ETX | DC3 | # | 3 | C | S | c | s |
| 4 | EOT | DC4 | $ | 4 | D | T | d | t |
| 5 | ENQ | NAK | % | 5 | E | U | e | u |
| 6 | ACK | SYN | & | 6 | F | V | f | v |
| 7 | BEL | ETB | ' | 7 | G | W | g | w |
| 8 | BS | CAN | ( | 8 | H | X | h | x |
| 9 | HT | EM | ) | 9 | I | Y | i | y |
| A | LF | SUB | * | : | J | Z | j | z |
| B | VT | ESC | + | ; | K | [ | k | { |
| C | FF | FS | , | < | L | \ | l | \| |
| D | CR | GS | - | = | M | ] | m | } |
| E | SO | RS | . | > | N | ^ | n | ~ |
| F | SI | US | / | ? | O | _ | o | DEL |

ASCII Control Codes

| | | | |
|---|---|---|---|
| ACK | Positive Acnowledge | BEL | Bell |
| BS | Backspace | CAN | Cancel |
| CR | Carriage Return | DCx | Device Control |
| DEL | Delete | DLE | Data Link Escape |
| EM | End of Medium | ENQ | Enquiry |
| EOT | End of Transmission | ESC | Escape |
| ETB | End of Transmission Block | ETX | End of Text |
| FF | Form Feed | FS | File Separator |
| GS | Group Separator | HT | Horizontal Tabulation |
| LF | Line Feed | NAK | Negative Acknowledge |
| NUL | Null | RS | Record Separator |
| SI | Shift In | SO | Shift Out |
| SOH | Start of Header | SP | Space |
| STX | Start of Text | SUB | Substitute |
| SYN | Synchronizing Character | US | Unit Separator |
| VT | Vertical Tabulation | | |

# APPENDIX E
# Dedicated Memory Addresses

| Address | Destination |
|---|---|
| FF00 | DRA and DDRA of PIA 1 |
| FF01 | CRA of PIA 1 |
| FF02 | DRB and DDRB of PIA 1 |
| FF03 | CRB of PIA 1 |
| FF20 | DRA and DDRA of PIA 2 |
| FF21 | CRA of PIA 2 |
| FF22 | DRB and DDRB of PIA 2 |
| FF23 | CRB of PIA 2 |
| FF40 | Disk control register |
| FF48 | Disk status (read)/command(write) |
| FF49 | Disk track # register |
| FF4A | Disk sector # register |
| FF4B | Disk data register |
| FFC0 - FFC5 | SAM video display mode |
| FFC6 - FFD3 | SAM video starting address |
| FFD4 - FFD5 | SAM Page # |
| FFD6 - FFD9 | MPU cycle rate |
| FFDA - FFDD | Memory size |
| FFDE - FFDF | Memory map mode |
| FFF2 - FFF3 | SWI3 vector location |
| FFF4 - FFF5 | SWI2 vector location |
| FFF6 - FFF7 | FIRQ vector location |
| FFF8 - FFF9 | IRQ vector location |
| FFFA - FFFB | SWI vector location |
| FFFC - FFFD | NMI vector location |
| FFFE - FFFF | RESET vector location |

# Index

[illegible]

With the right software the Color Computer is almost omnipotent. In your hands is a perfect example. This book was written on the Color Computer using the Telewriter-64 (Cognetic) word processing program. Also, every word of text was typeset using the CoCo and the Hewlett Packard LaserJet printer. The possible uses of the CoCo seem limitless.

This book shows in a tutorial fashion how to program in assembly language. Assembly language may be low level but it provides access to all the CoCo capabilities and programs written in this language execute very quickly. Therefore, all the hardware functions and capabilities of the CoCo and how to control them with assembly language are also explained in this book. A knowledge of assembly language will allow you to write your own unique programs. This knowledge will also allow you to determine how other programs work and then modify them to suit your needs.